

Checking and Certifying Computational Results

Jonathan D. Bright

A dissertation submitted to The Johns Hopkins University in conformity with the
requirement for the degree Doctor of Philosophy.

Baltimore, Maryland

1994

Abstract

For many years, there has been tremendous interest in methods to make computation more reliable. In this thesis, we explore various techniques that can be implemented in software to help insure the correctness of the output of a program. The basic tool we use is a generalization of the notion of a program checker called a certifier. A certifier is given intermediate computations from a program computing an answer in an effort to simplify the checking process. The certifier is constructed in such a way that even if the intermediate computations it is given are incorrect, the certifier will never accept an incorrect output.

We have constructed certifiers and program checkers for several common abstract data types including mergeable priority queues and splittable priority queues. We have also constructed a certifier for an abstract data type that allows approximate nearest neighbor queries to be performed efficiently. We have implemented and experimentally evaluated some of these algorithms. In the parallel domain, we have developed both general and problem specific techniques for certifying parallel computation. Lastly, we have formally proven correct a certifier for sorting, and have analyzed the advantages of using certifiers in conjunction with formal program verification techniques.

This work forms a thesis presented by Jonathan D. Bright to the faculty of the Department of Computer Science, at the Johns Hopkins University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy, under the supervision of Professor Gregory F. Sullivan.

Acknowledgements

I would like to thank my advisor, Gregory Sullivan, for giving me an excellent research topic for my thesis, and for vastly improving my writing skills during my stay at Hopkins. Also, I had the most incredibly interesting discussions with him on a wide variety of topics.

All of the faculty members at Hopkins were always supportive and helpful. But I should specifically mention Mike Goodrich, Simon Kasif, Gerry Masson, and Scott Smith. Nancy Scheeler, Tensie Wenderoth, and Pierre Joseph in the departmental office helped make my stay at Hopkins pleasant and disaster free.

I have known many students in the Department and in the University as a whole during my stay here. Each one had his or her own unique way of thinking about things and my friendships with them certainly broadened my horizons. But I would like to specifically mention Mujtaba Ghouse, Lewis Stiller, Paul Callahan, Dwight Wilson, Joanne Houlahan, Sarah Manchester and her roommates, Kumar Ramaiyer, and Pisupati Chandrasekhar for many interesting conversations and enjoyable times at Hopkins.

Last, but not least, I'd like to thank Mom and Dad.

Contents

Table of Contents	iv
List of Tables	vi
List of Figures	vii
1 Introduction	1
2 Checking Mergeable and Splittable Priority Queues	15
2.1 Preliminaries	16
2.1.1 Certifying Alphabetic Search Trees	18
2.2 Definitions	19
2.3 Checking Mergeable Priority Queues	20
2.3.1 A Theoretically Optimal Checker	20
2.3.2 A Practical Variant	30
2.4 Timing Results	34
2.5 Equivalence Between Checking MSTs and Sequences of MPQ Operations	36
2.6 Checking Splittable Priority Queues	40
3 On-Line Certification of Mergeable and Splittable Priority Queues	50
3.1 Certification of Search Data Structures	53
3.2 On-Line Checking of Mergeable Priority Queues	55
3.2.1 A First Phase Non-optimal Algorithm	56
3.2.2 A First Phase Optimal Algorithm	58
3.3 On-line Checking of Splittable Search Data Structures	66
4 On-Line Certification of Approximate Nearest Neighbor Queries	72
4.1 Calculating Approximate Nearest Neighbors	75
4.2 Checking The Correctness	75

5	Basic Parallel Techniques	80
5.1	Simulation Lemma	82
5.2	Parallel Sequence Evaluation Structure	87
5.2.1	Constructing and Checking the PSE Structure	92
5.2.2	Certifying Intersections of Isothetic Line Segments	98
5.2.3	Certifying 3-Dimensional Maxima	100
5.3	Other Common Parallel Problems	106
6	Parallel Checking and Certifying of Minimum Spanning Trees	111
6.1	Introduction	112
6.2	Definition of Prim Sequence	112
6.3	Checking on a Linear Array of Processors	116
6.4	PRAM Implementation	117
6.4.1	Finding the Prim Sequence of a Tree in Parallel	117
6.4.2	Evaluating the Concatenate-Merge Tree	120
6.4.3	Solving a Line Segment Query Problem	125
6.4.4	Constructing the Dominance Tree	127
6.4.5	Certifying Minimum Spanning Trees	128
7	Formally Verifying Certifier Programs	130
7.1	Advantages of Certification Trails	131
7.1.1	Software Development Advantages	131
7.1.2	Run-Time Advantages	132
7.2	The Boyer-Moore Theorem Prover	135
7.3	Definition of the Sorting Certifier	137
7.4	Experimental Results	141
7.5	Proofs of Soundness and Completeness	146
7.5.1	Proof That <i>certsort</i> is Sound	146
7.5.2	Completeness Result for <i>certsort</i>	149
8	Conclusion	156
	Bibliography	159

List of Tables

2.1	1 Priority Queue, N <i>Insert</i> followed by N <i>Deletemin</i> Operations . . .	35
2.2	1 Priority Queue, N <i>Insert</i> , 2N <i>Insert</i> and <i>Deletemin</i> , N <i>Deletemin</i> . .	35
2.3	200,000 Operations, 200 Expected Merges	36
2.4	400,000 Operations, 400 Expected Merges	36
7.1	Execution speed comparisons for Faster-CertSort, Allegro-Sort, and C-Sort.	145

List of Figures

1.1	Illustration of certification trail technique.	4
1.2	Illustration of the run-time advantage of certification trails.	6
2.1	Simple MPQ example.	21
2.2	More complex MPQ example.	22
2.3	Reduction of MST check to MPQ operation sequence check.	38
2.4	SPQ example.	42
3.1	Example of an array indexed set of doubly linked lists.	54
3.2	Example of tritter tree representation of MPQ.	57
3.3	Illustration of an H-heap.	60
3.4	Merging two H-heaps.	61
3.5	Representation of the splittable search data structure for the second phase.	67
4.1	Example of approximate nearest neighbor query.	74
4.2	Example of covering the box around the sphere.	77
5.1	Illustration of simulation.	84
5.2	Illustration of Parallel Sequence Evaluation structure	91
5.3	Decomposition of XY plane.	101
5.4	Illustrations for lemma 4.2.5.	103
5.5	Illustrations for lemma 4.2.6.	105
6.1	Construction of concatenate-merge tree from a weighted spanning tree.	119
6.2	Concatenate-merge tree T_L and dominance tree D_L	123
6.3	Concatenate-merge tree T_R and dominance tree D_R	123
6.4	Dominance tree for the concatenate-merge tree with $ $ at the root, T_L as the left subtree, and T_R as the right subtree.	124
6.5	Dominance tree for the concatenate-merge tree with \otimes at the root, T_L as the left subtree, and T_R as the right subtree.	124

7.1 Savings possible when using the certification-trail technique. 135

Chapter 1

Introduction

In this thesis we study the use of software in the role of making a system tolerant of hardware faults. We also study methods to make a system tolerant of software faults. Finally, we study methods of providing software which contains no faults in the first place. All of our methods are software based, though it is clear that most of our techniques can also be implemented using hardware, and we discuss this later.

Throughout most of this thesis we concentrate on detecting faults and assume that once a fault has been detected, other methods will be invoked to determine what to do about it. If the fault is determined to be a software fault, then we report that an error occurred, and we assume that a backup program (written in the spirit of N-version programming) is available to solve the problem. If the fault is a hardware fault, then there are two possibilities. If it is transient or intermittent, it may be sufficient to merely note that a fault occurred, and then recompute the answer. However, if the fault is permanent, then steps need to be taken to isolate the faulty component from the rest of the system.

The basic idea in all this work is the use of program checkers, and a generalization of the usual definition of program checkers to allow them to run more efficiently by using certification trails. A certification trail is composed of intermediate computational results created by the program computing the answer. Before describing our methodology in more detail, we now review some other software based fault tolerance techniques.

The most basic software based technique is simple time redundancy [51]. A program is run several times, and the outputs of all the runs are compared. If they agree, it is assumed that there were no faults. If they disagree, then an error is signaled. This method can be generalized in two obvious ways. First, if an error is detected, we could iterate this approach several times. If at any point all of the runs for the current iteration agree, then it is assumed that no faults occurred for the current iteration, and the answer is output. Second, we could use a voting mechanism to output the majority answer if not all of the runs agree. For example, if we ran the program three times, and one run faulted and produced an incorrect output, a voter would recognize that indeed two of the runs agreed, and output that result. Both of these generalizations transform the simple time redundancy approach from being a fault-detection method into a more broad fault tolerance method. In any case, the

time redundancy approach is most useful for detecting transient hardware faults. It is of limited use for detecting permanent hardware faults or software faults, though some interesting attempts have been made to extend this technique for these types of faults as well.

The next technique that we discuss is N-version programming [7][8][25]. Here, a specification for a problem is developed, and given to N independent programming teams. Each programming team writes a separate program for the problem. When we want the solution to some instance of the problem, we run each program on the input, and then check that all of the outputs are the same. If they all agree we output the answer, else we output that an error has occurred. Alternatively, we could use a voting mechanism, so that if there was a common answer that appeared as output from more than half of the programs, that common answer would be assumed to be correct. The N-version programming technique is designed to provide protection from software faults, although it clearly provides protection from hardware faults as well. Although intuitively it might appear to be very effective in detecting software faults, empirical studies have shown that even isolated programming teams can make coincident errors, and thus even if all of the programs produce the same output, it is not necessarily the case that the output is correct.

The next technique that we review is called algorithm-based fault tolerance [49][64]. At a high level, this technique uses error detecting and correcting codes for performing reliable computations with specific algorithms. This technique has proven to be very effective with matrix computations, and has also been adapted for other problems such as FFT computations. Generally, faults are detected with a high probability. If the number of faults that occur can be bounded by some small constant, then algorithm-based fault tolerance techniques will often be guaranteed to detect the error.

Another technique for software based fault tolerance is the use of program checkers. A program checker takes an input and the output of another program that solves the problem that we are interested in. If the output is correct the program checker outputs *ok*, else the program checker outputs *error*. This method is useful for detecting both software and hardware based faults, and can obviously be generalized in several different ways. First, the same checker can be run multiple times in the spirit of simple time redundancy, thereby increasing the transient hardware fault de-

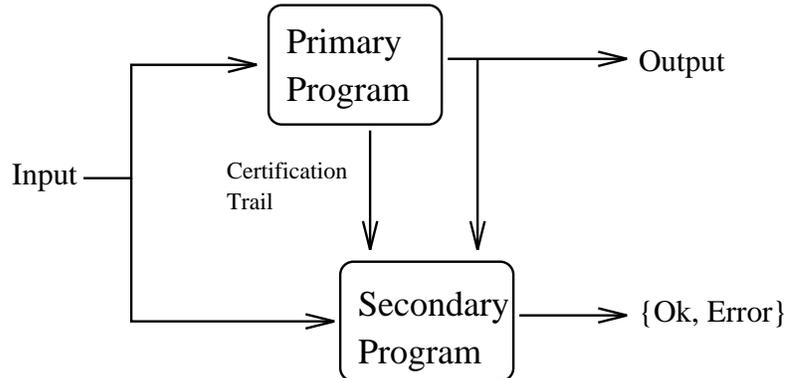


Figure 1.1: Illustration of certification trail technique.

tection capability. Second, multiple checkers can be written in the spirit of N-version programming to help increase the detection of software based faults. The main advantage that checkers have over simple time redundancy and the N-version programming method is that they typically require less resources at run time. And while they provide software based fault detection capabilities similar to that of N-version programming, since checkers typically have a simpler structure than a program which actually computes the answer, they are usually easier to develop and are more likely to be coded correctly. One would also expect there to be less chance of coincident errors in the checker and the program computing the answer as compared to 2-version programming, because the checker is performing a task conceptually different than the program computing the answer.

The concept of program checkers has a significant drawback though. There appear to be many problems for which it is difficult to develop efficient program checkers. However, it is possible to generalize the notion of a program checker to make it apply to more problems. One generalization, which is the basis of this thesis, is called the *certification trail* method. In the certification trail approach, programs F_1 and F_2 are developed for a problem. F_1 computes the answer, and also some additional data called a certification trail. F_2 takes the input, output, and the certification trail, and determines whether the output is correct. See Figure 1.1 for an illustration of the certification trail technique. The certification trail represents some intermediate computations made by F_1 , and enables F_2 to check the output easily. For example, if the input is supposed to be a permutation of the output, then the certification trail might be a mapping from the input to the output.

The certification trail technique has a close relationship to the “principle of the absolute supervisor.” We now quote a description of this important idea written by Jack Edmonds [33] in 1968 in the context of matroid partitioning.

We seek a good characterization of the minimum number of independent sets into which the columns of a matrix Π can be partitioned. As the criterion of “good” for the characterization we apply the “principle of the absolute supervisor.” The good characterization will describe certain information about the matrix which the supervisor can require his assistant to search out along with a minimum partition and which the supervisor can then use “with ease” to verify with mathematical certainty that the partition is indeed minimum. Having a good characterization does not mean necessarily that there is a good algorithm. The assistant might have to kill himself with work to find the information and the partition.

The “principle of the absolute supervisor” can be seen as the precursor of certification trails. The role of the first phase program is analogous to that of the assistant, and the role of the second phase program is analogous to that of the supervisor. The contribution of certification trails are twofold: (i) the application of the technique to new problem domains, (ii) discovery of *efficient* techniques to be used by the assistant to help convince the supervisor of the correctness of her work.

A formal description of the certification trail technique is given below. Note, this is a slight reformulation of the original certification trail definition [75] in which F_2 recomputed the output which was then compared to the output of F_1 instead of performing the comparison directly. In the definition we present here, if all of the faults that occur (whether they are hardware, or software, or a combination of both) are confined to one of the two program executions, then it is never the case that the second program accepts an incorrect output as correct.

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

Let $P : D \rightarrow S$ be a problem. Let T be the universal set. A solution to this problem using the *certification trail* method consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : D \rightarrow S \times T$ and $F_2 : D \times S \times T \rightarrow \{ok, error\}$. The functions must satisfy the following two properties:

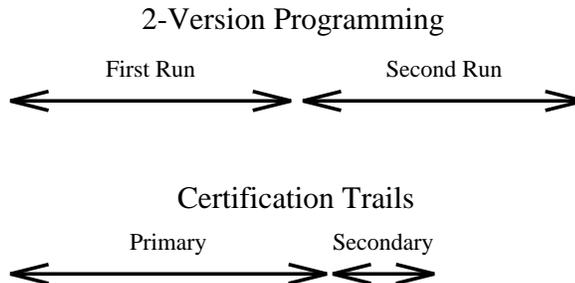


Figure 1.2: Illustration of the run-time advantage of certification trails.

1. For all $d \in D$ there exists $s \in S$ and there exists $t \in T$ such that $F_1(d) = (s, t)$ and $F_2(d, s, t) = ok$ and $(d, s) \in P$.
2. For all $d \in D$, for all $s \in S$, and for all $t \in T$, either $(F_2(d, s, t) = ok$ and $(d, s) \in P)$ or $F_2(d, s, t) = error$.

Note that the definition rules out the possibility that an unexpected certification trail can “fool” the checker into accepting an incorrect output. In the following discussions, we refer to the program F_1 as the first phase program, and F_2 as either the second phase or certifier program.

Whenever we design a certification trail solution to a problem, we want the asymptotic time complexity of the first phase to be the same as the asymptotic time complexity of the original algorithm, and the asymptotic time complexity of the second phase to be faster than the asymptotic time complexity of the original algorithm. But since we want a fault tolerance technique that is efficient in practice, we want a much more stringent condition to be satisfied. Specifically, we want the running time of the first phase plus the running time of the second phase to be less than twice the running time of the fastest algorithm which solves the problem. If this more stringent condition is satisfied, then the certification trail technique is guaranteed to have a run-time advantage over simple time redundancy and two-version programming. See Figure 1.2 for an example. In this figure, we see that the running time of the first phase is longer than the running time of the first execution for two-version programming. This is because of the overhead needed to produce the certification trail. However, the certification trail enables the second phase to run very quickly, and the total time taken by the certification trail technique is significantly

less than with two-version programming.

The certification trail technique has been applied to a wide variety of problems in computer science such as sorting, constructing convex hulls, finding minimum spanning trees, performing sequences of priority queue operations, constructing Huffman trees, finding the closest pair of points in a set, and many others [75][76][84][77]. The savings in running time for the second phase as compared to the second execution for two-version programming for these examples range from about 75% to about 90%. Also, for these examples the overhead needed to produce the certification trail is typically less than 5%.

The certification trail method has some similarities to the acceptance test method [68]. In the acceptance test method, after a computation has been finished and before the answer has been output, some checks are performed on the result to determine if an error has occurred. If an error is detected then various steps can be taken to remedy the situation. It is good practice to place an acceptance test at the end of a recovery block [68], so that if an error is discovered an alternate method for computing the answer can be attempted. However, there are also some key differences between the performance of these two methods. First, the certification trail method will sometimes allow for faster checking since the code performing the tests has the benefit of using a certification trail. Second, the certification trail method is sound. That is, if the certifier accepts the output then the output must be correct. The definition of an acceptance test allows for the possibility of an incorrect output not being caught by the test. Thus, in some cases acceptance tests can be faster than certification trails since acceptance tests do not have to detect all errors.

At this point, the reader may wish to see a specific example of a certification trail before proceeding further. In section one of chapter three, we present a certifier for the “ordered set” abstract data type, which is the abstract data type implemented by binary search trees. This would be a good example for a reader who is unfamiliar with the certification trail technique.

Before describing the specific problems that are addressed in this thesis, we describe a general technique that can be useful when attempting to construct a certification trail solution to a problem. Consider the situation when we have a program F that solves a problem P , and we would like to develop a certification trail solution F_1 and F_2 for P . If to solve P , the program F solves an instance of another

problem P' for which we have already developed a certification trail solution F'_1 and F'_2 , then it may be desirable to incorporate F'_1 and F'_2 into the programs F_1 and F_2 . We assume that F solves P' by calling the function $SolveP'$, and that $SolveP'$ only interacts with the program F through the arguments passed to it and the result returned. Specifically, we assume that $SolveP'$ neither reads from, nor writes to any global variables in F . We can construct F_1 and F_2 from F as follows:

- Let I be the input to F_1 . Construct F_1 from F by replacing the call to the procedure $SolveP'$ with a call to F'_1 . Let $I_{P'}$ be the instance of P' to be solved. Let C be the output of F'_1 . C is a pair of the form $(O_{P'}, C_{P'})$, and we use the first component of the pair as the solution to $I_{P'}$. At the last step, F_1 outputs a pair consisting of the answer O constructed, and the certification trail. The certification trail is C .
- Let the input to F_2 be the triple $(I, \tilde{O}, \tilde{C})$. Construct F_2 from F by replacing the call to $SolveP'$ with a call to F'_2 . Let $\tilde{C} = (\tilde{O}_{P'}, \tilde{C}_{P'})$. Let $I_{P'}$ be the problem instance of P' which is to be solved. F'_2 is given the triple $(I_{P'}, \tilde{O}_{P'}, \tilde{C}_{P'})$. We check that F'_2 returns *ok*. If so, use $\tilde{O}_{P'}$ as the solution to $I_{P'}$, and continue execution. At the last step compare the answer O constructed to \tilde{O} . If they are equal, output *ok*, else output *error*.

The following program fragments illustrate this idea. The variable C in F_1 is meant to be a variable name that is unused in F .

Procedure $F(I)$:

Begin

·

·

·

$O_{P'} = SolveP'(I_{P'})$

·

·

·

Output(O)

End

Procedure $F_1(I)$:

Begin

·
·
·

$C = F'_1(I_{P'})$

$O_{P'} = \text{first-component}(C)$

·
·

$\text{Output}(O, C)$

End

Procedure $F_2(I, \tilde{O}, \tilde{C})$:

Begin /* \tilde{C} is of the form $(\tilde{O}_{P'}, \tilde{C}_{P'})$ */

·
·
·

IF $F'_2(I_{P'}, \tilde{O}_{P'}, \tilde{C}_{P'}) = \text{error}$ THEN error

$O_{P'} = \tilde{O}_{P'}$

·
·

IF $(O \neq \tilde{O})$ THEN error ELSE ok

End

This scheme can obviously be generalized to handle the possibility that F solves a collection of different problem instances for a collection of different problems, where for each problem we have developed a certification trail solution. We call this generalized notion the fundamental certification trail principle, which we state as follows:

Principle 1.0.1: *Let P be a problem, and let F be a program which solves P . Assume that F , when given input of size n , spends $O(C(n))$ time in computation which is certifiable, and $O(T(n))$ time in computation which is not certifiable (or computation for which we do not want to apply certification trail techniques). Thus we see that F runs in $O(C(n) + T(n))$ time. Consider the computation which can be certified, and assume it can be certified so that the aggregate time for the first phase computations is $O(C_1(n))$, and the aggregate time for the second phase computations is $O(C_2(n))$. Then we see it is possible to construct a certification solution F_1 and F_2 for P so that F_1 runs in $O(C_1(n) + T(n))$ time, while F_2 runs in $O(C_2(n) + T(n))$ time.*

It is also possible to generalize this notion further so that it applies to on-line programs as well. An on-line program is given the input one operation at a time, and is expected to compute the answer for the current operation before the next operation is received. As an example, most transaction processing systems are on-line. An on-line certifier is given the input, output and certification trail one part at a time, and is expected to determine if there has been a failure in the first phase before the next input is received. Let F solve problem P in an on-line manner, and assume that to

solve P , F uses an on-line subroutine $SolveP'$ for a problem P' . That is, F makes repeated calls to $SolveP'$, and the output of F' for any particular call depends upon the previous invocations. Suppose that F'_1 and F'_2 form an on-line certification trail solution for P' . Then it is possible (using the same type of construction given above), to construct an on-line certification trail solution F_1 and F_2 for P using F'_1 and F'_2 as subroutines.

Commonly, the most important characteristics for an on-line system are the total time used to process an entire sequence of operations, and the worst-case incremental time needed to process any individual operation. The certification trail principle given earlier addresses the total time used for F_1 and F_2 , but we would also like a bound on the worst-case incremental time as well. For any program \mathbf{F} , let $inc-time(n, \mathbf{F})$ be the worst-case incremental time for \mathbf{F} over a sequence of n operations. Let the worst-case incremental running time of F be $O(T(n))$, excluding calls to $SolveP'$. Let $X(n)$ be the most number of calls to $SolveP'$ for any one individual operation while processing a sequence of size n . Thus we see that the worst-case incremental running time of F_1 and F_2 is $O(T(n) + X(n) * inc-time(X(n) * n, F'_1))$ and $O(T(n) + X(n) * inc-time(X(n) * n, F'_2))$ respectively. Specifically, if to process any one operation, F makes at most a constant number of calls to $SolveP'$, then the worst-case incremental time of F_1 and F_2 is $O(T(n) + inc-time(n, F'_1))$ and $O(T(n) + inc-time(n, F'_2))$ respectively.

In this thesis, we consider a variety of algorithms and abstract data types for which we present checkers and certifiers. We also explore the possibility of using formal verification methods on some of our checkers and certifiers. In particular, we consider the following problems:

- Off-line checking of mergeable priority queues. In previous work, Sullivan and Masson [76] presented a linear-time program checker for priority queues. We give a linear-time program checker for the generalized problem of checking mergeable priority queues. A mergeable priority queue implementation supports the existence of multiple priority queues, where any two of them can be destructively joined to form a single priority queue. By the term “off-line,” we mean that the checker is given both the input and output sequences in their entirety before it begins execution. We have implemented the program checker and have performed timing experiments on it. We show how this program checker can be used to certify the construction of alphabetic search trees.
- Off-line checking of splittable priority queues. A splittable priority queue im-

plementation supports multiple priority queues, where a given priority queue can be destructively split into two smaller priority queues according to a splitter element. All of the elements in the priority which are smaller than the splitter go into one of the resulting priority queues, and all of the elements that are greater than or equal to the splitter go into the other priority queue. We present a linear-time program checker for this problem as well.

- On-line certification of mergeable priority queues. Here, we present a certification trail solution for mergeable priority queues. The on-line certifier differs from the off-line checker because it is given the input, output and certification trail one operation at a time, and reports for each supposed answer in the output if it is correct before the next operation is received. The certifier runs in amortized time of $O(A(N))$ per operation where $A(N)$ is the inverse of Ackermann's function.
- On-line certification of splittable priority queues. In the spirit of the on-line mergeable priority certifier, we also present an on-line splittable priority queue certifier. The certifier runs in $O(1)$ time per operation.
- On-line certification of queries on an approximate nearest neighbor data structure. For an approximate nearest neighbor query, any point in the point set which is *almost* the nearest neighbor of the query point (where the value of "almost" is determined by an epsilon approximation factor) is considered an acceptable answer to the query. On a point set of size n , the first phase runs in $O(\log n)$ time per operation, and the second phase runs in $O(1)$ time per operation.
- Basic certification techniques for parallel algorithms. In this chapter we present a simulation lemma which shows how any parallel program (provided it executes in an appropriate model of parallel computation) can be certified so that the second phase runs in $O(1)$ time while performing the same total work as the first phase. Also, we present a certification trail solution for the problem of evaluating a sequence of set manipulation operations, and give several applications of this result.
- The formal verification of a sorting certifier. The formal verification of the program was performed using the Boyer-Moore theorem prover. We analyze the potential advantages of formally verifying checker/certifier programs, and in addition perform timing experiments on the formally verified sorting certifier program.

Of all of the other software based fault tolerance methods, program checkers and probabilistic program result checkers (which we describe shortly), achieve results

that are closest in spirit to certification trails. However, these other methods treat the program being checked as a black box, as opposed to the certification trail technique where the program being checked is intentionally modified. While treating the program being checked as a black box gives these methods a desirable generality (i.e. a program checker for a problem can work with any program which computes the answer), it also potentially can limit the applicability and practical effectiveness of these methods.

Now we review some results in the program checking model. The minimum spanning tree property of a tree in a graph was shown to be checkable in $O(nA(n)+m)$ time, where n is the number of vertices in the graph, m is the number of edges in the graph, and $A(n)$ is the inverse of Ackermann's function [79]. Later, this result was improved to $O(n+m)$, and this new algorithm is efficiently parallelizable [31]. Even more recently, it was shown that a minimum spanning tree of a graph could be found in randomized expected linear time [54]. Checking the correctness of a sequence of n priority queue operations was shown to take $O(n)$ time [76]. In the same paper, a simple $O(n)$ time algorithm was presented which could check a sequence of n union-find operations, though more sophisticated theoretical algorithms already existed for this problem. An $O(n)$ algorithm was developed which could check the correctness of either extremely skewed or extremely well balanced alphabetic search trees [60]. It is still an open problem whether more general alphabetic search trees can be checked in $O(n)$ time.

We now review a model related to the program checking model called the probabilistic program result checking model [14]. In the probabilistic program result checking model we are given a program P which supposedly solves a problem. We wish to construct a program P' which solves the same problem, but in a more reliable manner. P' is allowed to make calls to P , and the *incremental* running time of P' excludes the time taken when P is invoked by P' . We assume that P' executes flawlessly, but that P can contain software bugs. Given a reliability parameter k , we want P' to output an incorrect answer with probability less than or equal to $1/2^k$. If a correct answer can not be produced because P contains a software bug, then the keyword BUGGY is output. Otherwise, the correct answer is output. By constraining the incremental running time of P' to be less than the running time of P , we can force P' to perform the computation in a markedly different manner than P . The general

idea is that on successive calls to P , the input/output pairs generated can be checked against each other. A probabilistic program result checker P' has no “worst case” input I which always causes incorrect behavior of P' , and this is regardless of the types of faults experienced by P . This is an extremely useful property to have when it is hard to know ahead of time the distribution of inputs that will be seen at run time. A probabilistic program result checker P' which detects errors with probability one, and only submits the original input I to the program P , is by definition a program checker.

In the probabilistic program result checker model, other interesting results have been obtained [14]. Suppose we are given a program P which supposedly determines if two graphs G and H are isomorphic. It is possible to construct a program P' so that if P' reports that G and H are isomorphic, then this will be true with probability one, if P' reports that G and H are nonisomorphic, then this will be true with probability $1 - 1/2^k$, where k is any positive integer. k is given as input to P' , and P' performs k calls to P . If P has a software bug, then P' is allowed to output BUGGY if it is unable to determine if G and H are isomorphic or not. Probabilistic program checkers in the same spirit are also constructed for several important group theoretic problems, and also for determining the rank of a matrix.

The same paper also presents two *program checkers* for sorting, both of which operate in a probabilistic manner. Given a correct input/output pair, each checker will always output correct. Given an incorrect input/output pair, then with very high probability the checkers will output an error.

Another paper in the area of probabilistic program result checking concerns the checking of random access memories, and some simple data structures such as stacks, queues, trees, and graphs [13] [3]. The strong point of these algorithms is that they need only a small internal memory to monitor a large external memory or data structure. For example, the checker may only need an internal memory of size $O(\log n)$ while checking a data structure of size $O(n)$. However, since operations on these data structures take only $O(1)$ time per access, it is impossible to obtain an asymptotic speed up for the checker program. In fact, some of checkers presented require $O(\log n)$ time to check each operation.

The paper [15] presents results in the self-testing/correcting model, which is related to the probabilistic program result checking model. Given a program F for a

problem, these results allow one to estimate the percentage of inputs for which F computes the correct answer. Also, assuming that F is not too faulty on average, these results allow one to compute the correct output for *any* specific input with high probability. They provide general techniques for a variety of numerical problems including matrix multiplication, computing determinants, and polynomial multiplication.

There are two other related theoretical models which we now discuss. The paper [11] allows for checking that a proof of length $O(n)$ is correct in only polylogarithmic time. An error correcting code is applied to the original proof, which increases the length of the proof to $O(n^{1+\epsilon})$. The new proof needs to be sampled by the checker in only a polylogarithmic number of locations to make sure that it is correct with a high probability. While the checker runs quickly, having to apply the error detecting code to the original proof currently makes the technique prohibitive in practice. The final technique that we review are zero knowledge proofs [42][34]. These techniques allow one party to convince a second party about the truth of a theorem without giving the second party any information which would aid it in constructing a proof itself. Though zero knowledge proofs are not specifically intended for the problem of checking the output of a program, much of the work on probabilistic program result checkers emerged from earlier research on zero knowledge proofs.

Chapter 2

Checking Mergeable and Splittable Priority Queues

2.1 Preliminaries

This chapter concerns three fundamental abstract data types: priority queues (PQs), mergeable priority queues (MPQs) and splittable priority queues (SPQs). These abstract data types have been recognized as centrally important from the early days of computer algorithm design. They appear in seminal algorithm texts such as Knuth's [56] and Aho, Hopcroft and Ullman's [1]. Data structure implementations for these abstract data types continue to proliferate and currently include: binomial queues, AVL trees, leftist trees, skew heaps, pairing heaps, Fibonacci heaps, 2-3 trees, b-trees and others.

We present algorithms designed to check the answers generated by these abstract data types. To evaluate a sequence of N priority queue operations requires $\Omega(N \log N)$ time. (This lower bound uses a decision tree model of computation.) Since MPQs and SPQs are generalized versions of priority queues they also require $\Omega(N \log N)$ time to evaluate N operations. However, we have designed $O(N)$ time algorithms which check the answers given by these abstract data types. These linear time answer checkers are the first known for the MPQ and SPQ abstract data types. The paper [76] presents a linear time algorithm for checking the answers from PQs. We have found that the MPQ algorithm can also be used as a new efficient linear time checker for PQs.

This chapter also contains an experimental evaluation of our MPQ checker being used to check both MPQs and PQs. In order to evaluate the relative speed and effectiveness of our MPQ checker, we obtained the code to five different PQ and MPQ implementations which were recently tested in an article which appeared in the *Application Experience* section of the Journal *Algorithmica* [62]. This study is an extensive evaluation of different PQ and MPQ implementations. Our results show that the checker is substantially faster than the fastest of these implementations of PQs and MPQs on a wide variety of operation sequences.

The checkers we have designed have several potential applications in error monitoring, software debugging, repairable data structures and certification trails which we briefly discuss now. The desirability of using software to monitor a computation to attempt to determine if it is executing properly has long been recognized in the fault-tolerance community. For example, the general approach designated algorithm-based fault tolerance [49] uses software and additional data to attempt to

detect errors. That approach is particularly valuable for detecting errors in fundamental operations such as matrix multiplication and FFT. Our checkers provide an error detection capability for other fundamental operations, viz., PQ, MPQ and SPQ operations.

Our checkers can also be used as acceptance tests in the recovery-block approach [68, 6] to software fault tolerance. After an error is detected many methods of recovery are possible. For example, the data contents of the abstract data type might be extracted and placed in a new abstract data type which is based on an alternative implementation strategy. Alternatively, the old data contents might be abandoned and the previous abstract data type operations might be replayed for a new implementation. A third method could be based on the use of a repairable data structure as discussed further below.

Note, our checkers are based on 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, we have been able to design more efficient checkers. In the section on algorithm implementation we discuss methods for reducing this latency. In the next chapter, we present certification trail solutions for these abstract data types which have no latency. However, the efficiency of the second phase program for mergeable priority queues is slightly reduced.

A vivid example of the value of our checkers in the software development and debugging process was provided when we performed our experimental timing studies. To conduct our experiments we first searched for preexisting PQ and MPQ implementations. The author of [62] graciously provided us with copies of his extensively benchmarked and tested implementations to use in our studies. We generated many randomized sequences of operations and used our MPQ checker to check each of the five implementations provided. This methodology allowed us to discover a bug in the implementation of binomial queues.

In general, we believe checkers such as those presented here can complement the software development process by aiding in the detection of bugs. Checkers appear to allow superior exploitation of design diversity since they can utilize different algorithms which may be less likely to contain coincident errors than a reimplementa-

of the abstract data type being tested. Also, checkers can often be significantly faster than the code being checked and speedy checkers allow one to perform tests on a large corpus of data.

Another possible application for our checkers occurs when one is used in conjunction with a repairable data structure which allows for repair but does not automatically attempt to detect faults [81]. Suppose a PQ, MPQ or SPQ abstract data type is implemented with a repairable data structure. One can use one of our checkers 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.

2.1.1 Certifying Alphabetic Search Trees

In this section, we discuss a general technique for using checkers to generate certification trails. Consider an algorithm which uses a particular abstract data type, and assume there exists an efficient checker for the abstract data type. Using the fundamental certification trail principle, we could use the checker to help construct an efficient certification trail solution for the algorithm. We modify the algorithm so that whenever an operation on the abstract data type is performed, the answer generated is output as part of the certification trail. During the second phase execution, instead of maintaining the abstract data type, the algorithm reads the supposed answers from the certification trail for all of the query operations. As the last step, the algorithm runs the checker program on the sequence of operations generated and on the supposed answers in the certification trail to verify that the supposed answers are correct.

As a specific example, our mergeable priority queue checker can be used to certify the problem of constructing alphabetic search trees. Suppose we are given a set of keys $k_1 \leq k_2 \leq \dots \leq k_n$. Assume each key k_i has weight w_i , where the weight w_i represents a frequency with which the key k_i is referenced. The problem is to build a binary search tree on the keys so that the sum $\sum w_i l_i$ is minimized, where l_i is the depth of the key k_i in the tree. Intuitively, keys with large weights tend to be near the root of the tree.

Knuth [57] shows how the Hu-Tucker algorithm [48] for constructing alphabetic search trees can be implemented so that $O(n \log n)$ time is spent performing

mergeable priority queue operations, and only $O(n)$ additional time is spent performing other computations. Using the scheme discussed above and the mergeable priority queue checker that we present in the following sections, we see that we can certify the problem of constructing alphabetic search trees so that the second phase runs in $O(n)$ time. Previously, it was shown how either extremely skewed or extremely well balanced alphabetic search trees can be checked in linear time [60], though it is still an open problem whether arbitrary alphabetic search trees can be checked in linear time.

2.2 Definitions

We now describe the operations that we consider. The basic operand type is a pair of the form (it, val) . it is a positive integer, and is called the item number of the pair. Item numbers are used for referring to pairs, and hence all pairs must have distinct item numbers. val can be an arbitrary real number, and is called the item value of the pair. Item values are used to determine the relative order of the pairs. In the case when two pairs have the same item value, we use the item numbers of the pairs to break ties. Thus, we have the total ordering $(it, val) < (it', val')$ iff $val < val'$ or $(val = val'$ and $it < it')$. On occasion we refer to *pairs* as *elements*. Sets are referred to by a set name, which is assumed to be a positive integer.

1. $makeset(S)$: Create the set S and initialize it to be empty. S must not have been used previously in the sequence.
2. $insert((it, val), S)$: Insert the pair (it, val) into the set S . Set S must exist, and item number it must not have been used previously.
3. $delete(it)$: Delete the pair with item number it from whatever set it is currently in. Some pair with item number it must be in some set. When we discuss how to check splittable priority queues, we assume that $delete(it)$ returns the name of the set that the pair with item number it was contained in.
4. $min(S)$: Return the pair with smallest value that is in set S . Set S must exist and may not be empty.
5. $deletemin(S)$: Return the pair with smallest value that is in set S , and then delete it from set S . Set S must exist and may not be empty.

6. $merge(S_1, S_2, S)$: Combine the contents of sets S_1 and S_2 to form set S . Set S_1 and set S_2 must exist, while set S may not have been used previously in the sequence. After this operation sets S_1 and S_2 are destroyed.
7. $split(S, val, S_1, S_2)$: Form the set S_1 by taking all of the pairs in S which are of value smaller than or equal to (∞, val) , and form the set S_2 by taking all of the pairs in S which are of value greater than (∞, val) . Either S_1 or S_2 may be empty. Set S must exist, and sets S_1 and S_2 must not have been used previously in the sequence. After this operation set S is destroyed.

Note that the minimum value of a set is always unique, since we use the item numbers of pairs for secondary comparisons. All of the operations which do not return pairs are assumed to return the value *nil*. Also, the requirement that set names and item numbers not be reused is solely for the ease of describing our algorithms. In all cases it is not difficult to modify our algorithms so that item numbers and set names can be reused.

A data structure that supports the operations *makeset*, *insert*, *delete*, *min*, and *deletemin* we call a priority queue. A data structure that supports priority queue operations, and in addition supports the *merge* operation we call a mergeable priority queue. A data structure that supports priority queue operations, and in addition supports the *split* operation we call a splittable priority queue. Sometimes throughout the chapter we abbreviate *deletemin* operations as *dmin*.

2.3 Checking Mergeable Priority Queues

Let $O = (o_1, o_2, \dots, o_n)$ be a sequence of MPQ operations. Let $A = (a_1, a_2, \dots, a_n)$ be a sequence such that for all i , a_i is of the form (it, val) if o_i is a *min* or *deletemin* operation, and a_i is *nil* otherwise. We wish to determine if A contains the correct answers to the *min* and *deletemin* operations in O .

2.3.1 A Theoretically Optimal Checker

In this section we will present an algorithm to check the correctness of a sequence of MPQ operations. Our algorithm processes the operations in the sequence one by one. Based upon the *makeset*, *min*, *deletemin*, and *merge* operations in O , it constructs a forest of weighted trees. The tree edges are maintained in the set *treeedges*. Each

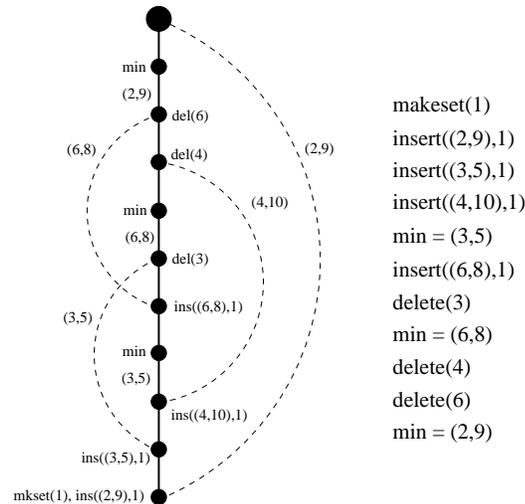
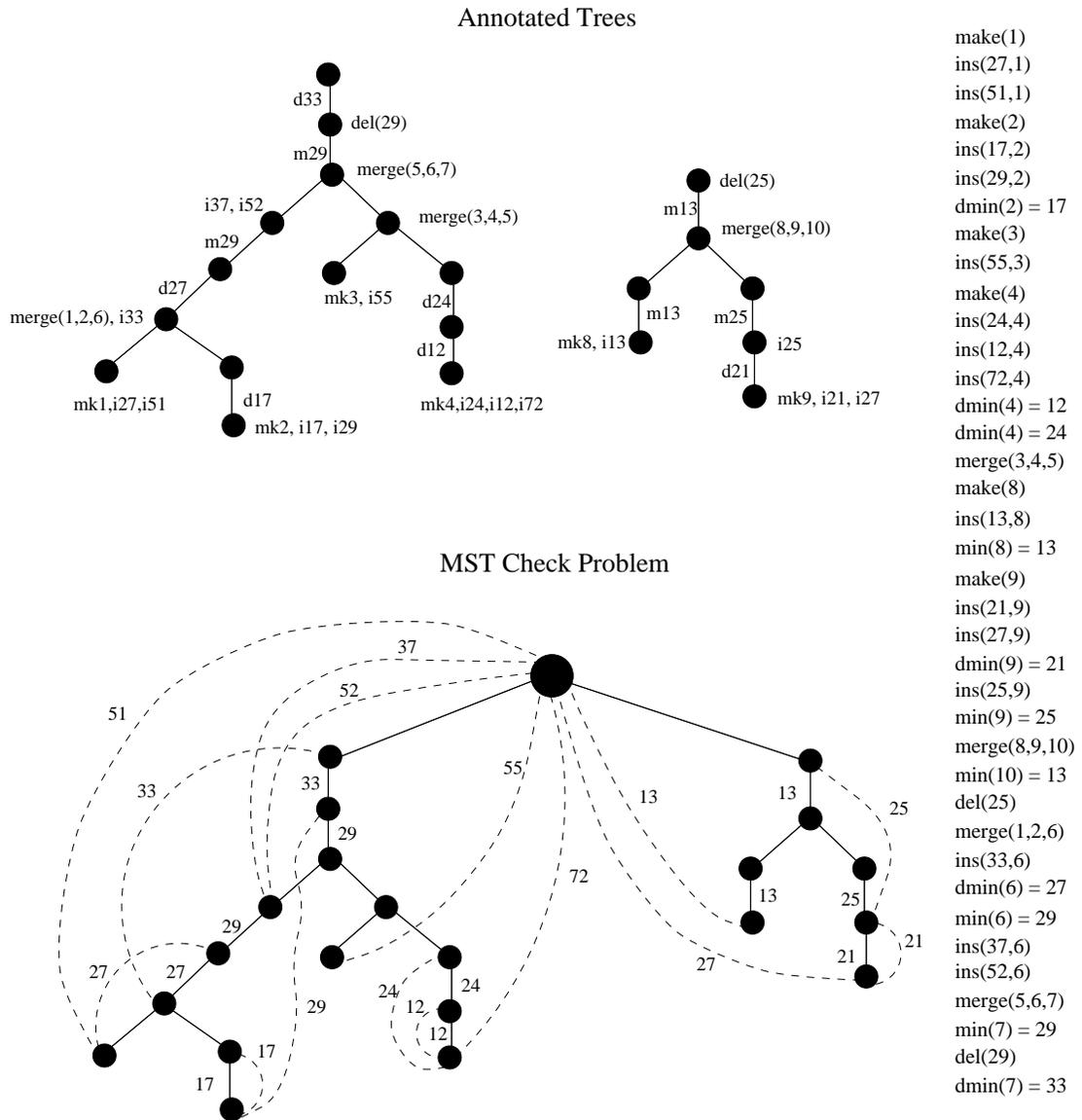


Figure 2.1: Simple MPQ example.

tree of the forest has a root which is kept track of by the function *curroot* (an abbreviation for current root). During any stage of the algorithm, there is a one-to-one correspondence between the active sets (i.e., sets which have been created and not yet destroyed) and the trees of the forest. This correspondence is maintained by *curroot*. During execution the algorithm also maintains a set called *checkedges*. This set contains edges which interconnect nodes within trees. The placement of these edges is carefully based on when and where elements are inserted into and deleted from the MPQs. These edges allow the algorithm to check if the answer to a *min* operation truly is the minimum element of a set and these edges allow other similar checks.

In the final phase of the algorithm a single weighted tree is created from the edges in *treeedges* by connecting all the roots in the forest to a single new root. A graph is also constructed based on the edges in $treeedges \cup checkedges$. If the tree is a minimum weighted spanning tree of the graph then the answers given for the operations are correct, otherwise the answers are incorrect. A note needs to be made here. Specifically, we require a generalized definition of Minimum Spanning Trees. Typically, the definition of MSTs requires that an addition operator be defined on the weights of the edges. However, it is possible to define MSTs even when no addition operation is given. This is desirable when the operands to the mergeable priority queue operations are totally ordered but do not have an addition operation defined

Figure 2.2: More complex MPQ example.



on them. We use the definition below which is a generalization of the standard minimum spanning tree definition, see [79].

Definition 2.3.1: Let G be a graph with vertices V and edges E . Also, let G have a weight function defined on E . Let $T \subseteq E$ be a spanning tree of G . We say that T is a minimum weight spanning tree (MST) of G iff for every $(u, v) \in E - T$, $weight(u, v)$ is greater than or equal to the $weight$ of any edge on the unique simple path from u to v in T .

See Figure 2.1 for a simple example involving only one priority queue with *insert*, *delete*, and *min* operations. Edges in *treeedges* are black and edges in *checkedges* are dashed. The ordered pairs are the edge weights. The other labels are annotations which help guide the reader in understanding the algorithm's execution. See Figure 2.2 for a more complex example. We use the abbreviations *make*, *ins*, *del*, and *dmin* for *makeset*, *insert*, *delete*, and *deletemin* respectively. To simplify the presentation of the more complex example we have omitted the item numbers of the elements. Thus for the *delete* operation we have used an item value to indicate which element should be deleted. In actuality, the item number of the element would be used. The top part of the figure represents only the forest constructed based on *treeedges* after the last operation is processed and before the final stage. Each leaf node has a label "mk" corresponding to the *makeset* operation which created it. Some leaf and internal nodes are labeled with "i" corresponding to the *insert* operations on the sets. Some internal nodes are labeled with *merge* corresponding to *merge* operations on the sets. Some edges are labeled with "d" or "m" corresponding to the *deletemin* and *min* operations on the sets. The bottom part of the figure is the final MST check problem. All unlabeled edges are assumed to have weight $(0, -\infty)$.

We now present algorithm *CheckMPQ* which takes as input a sequence O of operations and a sequence A of supposed answers. During the initial reading of this algorithm, it is best to ignore the IF/THEN statements in the Main Loop, and to concentrate on the rest of the algorithm which constructs the trees and the graph. The IF/THEN statements in the Main Loop detect errors relating to the input sequence and supposed answers not being partially correct (which is formally defined after the algorithm is presented).

Procedure *CheckMPQ*(O, A):

Let \mathbf{S} be the set of all set names appearing in O .
 Let IT be the set of all item numbers appearing in O .
 Let \mathbf{R} be the universe of item values.
 Let Γ be a set of graph nodes.
 Let $checkedges$ be a subset of $\Gamma \times \Gamma$ which is initially empty.
 Let $treeedges$ be a subset of $\Gamma \times \Gamma$ which is initially empty.
 Let $weight$ be a function from $checkedges \cup treeedges$ to $IT \times \mathbf{R}$.
 Let $curroot$ be a function from \mathbf{S} to $\Gamma \cup \{nil, destroyed\}$.
 Let $whereins$ be a function from IT to $\Gamma \cup \{nil, done\}$.
 Let $value$ be a function from IT to \mathbf{R} .
 For each $S \in \mathbf{S}$ initialize $curroot(S)$ to be nil .
 For each $it \in IT$ initialize $whereins(it)$ to be nil .

The input is a sequence of N operations and supposed answers: O and A .
 If there is an error then routine $error$ is called otherwise the answers are correct
 and the routine ok is called.
 For each i between 1 and N execute the appropriate case depending on the i th operation:

Main loop:

$makeset(S)$:

IF ($curroot(S) \neq nil$) THEN $error$.
 Let r be an unused node from Γ and set $curroot(S) = r$.

$insert((it, val), S)$:

IF ($curroot(S) = nil$) OR ($curroot(S) = destroyed$) OR ($whereins(it) \neq nil$) THEN $error$.
 Set $whereins(it) = curroot(S)$, and $value(it) = val$.

$delete(it)$:

IF ($whereins(it) = nil$) OR ($whereins(it) = done$) THEN $error$.
 Let S be the set which contains the pair with item number it .
 Let $v_1 = whereins(it)$, and let $v_2 = curroot(S)$.
 Add the edge (v_1, v_2) to the set $checkedges$, and let $weight(v_1, v_2) = (it, value(it))$.
 Set $whereins(it) = done$.

$min(S)$:

Let (it, val) be the supposed answer.
 IF ($whereins(it) = nil$) OR ($whereins(it) = done$) THEN $error$.
 IF ($value(it) \neq val$) THEN $error$.
 IF ($curroot(S) = nil$) OR ($curroot(S) = destroyed$) THEN $error$.
 IF set S does not contain (it, val) THEN $error$.
 Let r be an unused node from Γ .

Add $(r, \text{curroot}(S))$ to *treeedges* and let $\text{weight}(r, \text{curroot}(S)) = (it, val)$.
 Set $\text{curroot}(S) = r$.

deletemin(S):

Let (it, val) be the supposed answer.
 IF $(\text{whereins}(it) = \text{nil})$ OR $(\text{whereins}(it) = \text{done})$ THEN *error*.
 IF $(\text{value}(it) \neq val)$ THEN *error*.
 IF $(\text{curroot}(S) = \text{nil})$ OR $(\text{curroot}(S) = \text{destroyed})$ THEN *error*.
 IF set S does not contain (it, val) THEN *error*.
 Let r be an unused node from Γ .
 Add $(r, \text{curroot}(S))$ to *treeedges* and let $\text{weight}(r, \text{curroot}(S)) = (it, val)$.
 Set $\text{curroot}(S) = r$, and let $v = \text{whereins}(it)$.
 Add the edge (v, r) to the set *checkedges*, and let $\text{weight}(v, r) = (it, val)$.
 Set $\text{whereins}(it) = \text{done}$.

merge(S_1, S_2, S):

IF $(\text{curroot}(S_1) = \text{nil})$ OR $(\text{curroot}(S_1) = \text{destroyed})$ THEN *error*.
 IF $(\text{curroot}(S_2) = \text{nil})$ OR $(\text{curroot}(S_2) = \text{destroyed})$ THEN *error*.
 IF $(\text{curroot}(S) \neq \text{nil})$ THEN *error*.
 Let r be an unused node from Γ .
 Add the edges $(r, \text{curroot}(S_1))$ and $(r, \text{curroot}(S_2))$ to *treeedges*.
 Let $\text{weight}(r, \text{curroot}(S_1)) = (0, -\infty)$ and let $\text{weight}(r, \text{curroot}(S_2)) = (0, -\infty)$.
 Set $\text{curroot}(S)$ to be r
 Set $\text{curroot}(S_1)$ and $\text{curroot}(S_2)$ equal to *destroyed*.

Final stage:

Let R be an unused node from Γ .
 For each set S such that $\text{curroot}(S) \neq \text{nil}$ and $\text{curroot}(S) \neq \text{done}$,
 add $(R, \text{curroot}(S))$ to *treeedges* and set $\text{weight}(R, \text{curroot}(S)) = (0, -\infty)$.
 Let P be the set of all pairs $(it, \text{value}(it))$ such that $\text{whereins}(it) \neq \text{nil}$
 and $\text{whereins}(it) \neq \text{done}$.
 For each pair $(it, \text{value}(it))$ in P add the edge $(\text{whereins}(it), R)$ to *checkedges*,
 and set $\text{weight}(\text{whereins}(it), R) = (it, \text{value}(it))$.
 Let Γ^* refer to the nodes actually used by the algorithm.
 Let G be the graph formed by Γ^* and the edges in *treeedges* \cup *checkedges*.
 IF the tree formed by *treeedges* is a minimum weight spanning tree of G THEN *ok*
 ELSE *error*.

CheckMPQ performs many simple checks. For example, it checks that neither a set name nor an item number are reused. It also checks that the argument of a

delete operation is actually present in some set when the *delete* operation is performed. It also checks that the key value associated with an item number is the same when that element is inserted and later deleted. We wish to discuss these relatively simple types of checks separately from the more difficult checks that CheckMPQ performs. Therefore we introduce the notion of partial correctness for a sequence of operations and answers.

*Let O and A be a sequence of operations and supposed answers for a mergeable priority queue. We say O and A are partially correct if the answers given for the operations are correct when the operations for *min* and *deletemin* are redefined as follows: $\text{min}(S)$ returns an arbitrary pair from the set S , and $\text{deletemin}(S)$ returns an arbitrary pair from the set S and also deletes it.*

It is not difficult to design a program to check for partial correctness. Indeed, the code for CheckMPQ contains all the necessary checks to determine if a sequence of operations and supposed answers is partially correct. (See below for implementation details.) It is not hard to show that if a sequence of operations and answers is not partially correct then CheckMPQ will indicate *error*. Also, if CheckMPQ indicates *error* in the main loop then the sequence was not partially correct.

Lemma 2.3.2: *Algorithm CheckMPQ is correct.*

Proof: It is not difficult to show that CheckMPQ behaves correctly for sequences which are not partially correct. Thus, we will assume that the input sequences are partially correct. This means the algorithm will not stop in the main loop with a call to *error* but will always execute the final stage.

A simple induction on the number of executions of the main loop can be used to prove the loop invariants given next. If $\text{whereins}(it) = \text{nil}$ then item it has not been inserted yet. If $\text{whereins}(it) = \text{done}$ then item it has been inserted and deleted. Otherwise, item it has been inserted but not yet deleted. If $\text{curroot}(S) = \text{nil}$ then set S has not been created by *makeset* yet. If $\text{curroot}(S) = \text{destroyed}$ then set S has been created by *makeset* and destroyed by *merge*. Otherwise, S has been created by *makeset* but has not been destroyed yet. Let us call such a set *active*.

Let us use Γ' to refer to nodes which have actually been used by the algorithm at some point during execution. The following invariants can also be easily proven. During each iteration of main loop, the edge set *treeedges* and the vertex set

Γ' form a forest. (By our convention an isolated node is a tree in this forest.) Further, there is a one-to-one correspondence between the individual trees of this forest and the active sets. This correspondence is given by the function $curroot$, i.e., $curroot(S)$ is a node which we call the root of the corresponding tree. Further, if a pair (it, val) is in a set S then $whereins(it)$ equals a node in the tree with root $curroot(S)$.

For our main proof we use a different induction. We assume the behavior of CheckMPQ is correct for all sequences of length $k - 1$ and we attempt to show it is correct for all sequences of length k . The base case for the induction is trivially true.

Let $treeedges_{k-1}$ be the value of $treeedges$ when the algorithm completes execution after being given only the first $k - 1$ operations and answers as input. Also, let $treeedges_k$ be the value of $treeedges$ when the algorithm completes execution after being given only the first k operations and answers as input. Similarly, we use the subscripts k and $k - 1$ on other sets and functions with the same meaning as above. Let G_k refer to the graph with vertices Γ_k and edges $treeedges_k \cup checkedges_k$. Let G_{k-1} refer to the graph with vertices Γ_{k-1} and edges $treeedges_{k-1} \cup checkedges_{k-1}$.

Recall, we are assuming that the operations and answers are partially correct and thus the final phase is always executed. Since MPQcheck behaves correctly on sequences of length $k - 1$ we know by examination of the final test performed in the code for MPQcheck that the first $k - 1$ operations and answers are correct iff $treeedges_{k-1}$ is an MST of the graph G_{k-1} . To complete our proof we must show that the first k operations and answers are correct iff $treeedges_k$ is an MST of the graph G_k .

To show this we perform a case analysis for each possible k th operation.

Case 1: Suppose the k th operation is $min(S)$ and the supposed answer is (it, val) . Note $treeedges_{k-1}$ and $treeedges_k$ are nearly identical. $treeedges_k$ has one additional edge $(curroot_k(S), curroot_{k-1}(S))$ which has weight (it, val) . In addition, the edge $(R, curroot_{k-1}(S))$ is replaced with the edge $(R, curroot_k(S))$. The weight $(0, -\infty)$ remains the same. Also, the sets $checkedges_{k-1}$ and $checkedges_k$ are identical.

Suppose that $treeedges_k$ is an MST of G_k . By the generalized MST definition and the observations immediately above we may conclude that $treeedges_{k-1}$ is an MST of G_{k-1} . This means that the first $k - 1$ operations and answers are correct. Note, that for every pair $(it', val') \in S_k$ there is an edge $(R, whereins_k(it'))$ with weight (it', val') in G_k . Note, the edge $(curroot_k(S), curroot_{k-1}(S))$ which has

weight (it, val) is on the simple path from R to $whereins_k(it')$ in $treeedges_k$ for every pair $(it', val') \in S_k$. By the generalized MST definition it follows that for every $(it', val') \in S_k$, $(it', val') \geq (it, val)$. Since the sequence is partially correct $(it, val) \in S_{k-1}$. By the definition of the *min* operation we conclude that the first k operations and answers are correct.

Now, suppose that $treeedges_k$ is not an MST of G_k . By the generalized MST definition it is not hard to see that either $treeedges_{k-1}$ is not an MST of G_{k-1} or there exists an edge of the form $(R, whereins_k(it'))$ with weight (it', val') such that $(it', val') < (it, val)$ and $(it', val') \in S_k$. In either case the sequence of k operations and answers is incorrect. This completes case 1.

Case 2: Suppose the k th operation is *deletemin*(S) and the supposed answer is (it, val) . Note, $treeedges_{k-1}$ and $treeedges_k$ are nearly identical. $treeedges_k$ has one additional edge which is $(curroot_k(S), curroot_{k-1}(S))$ which has weight (it, val) . In addition, the edge $(R, curroot_{k-1}(S))$ is replaced with the edge $(R, curroot_k(S))$. The weight $(0, -\infty)$ remains the same. Also, the sets $checkedges_{k-1}$ and $checkedges_k$ are nearly identical. $checkedges_k$ has the edge $(whereins_{k-1}(it), R)$ replaced with $(whereins_{k-1}(it), curroot_k(S))$. The weight of the edge, (it, val) , is unchanged.

Suppose that $treeedges_k$ is an MST of G_k . By the generalized MST definition and the observations immediately above we may conclude that $treeedges_{k-1}$ is an MST of G_{k-1} . This means that the first $k-1$ operations and answers are correct. Note, that for every pair $(it', val') \in S_k$ there is an edge $(R, whereins_k(it'))$ with weight (it', val') in G_k . By the generalized MST definition it follows that for every $(it', val') \in S_k$, $(it', val') \geq (it, val)$. Since the sequence is partially correct $(it, val) \in S_{k-1}$. By the definition of the *deletemin* operation we conclude that the first k operations and answers are correct.

Now, suppose that $treeedges_k$ is not an MST of G_k . By the generalized MST definition it is not hard to see that either $treeedges_{k-1}$ is not an MST of G_{k-1} or there exists an edge of the form $(R, whereins_k(it'))$ with weight (it', val') such that $(it', val') < (it, val)$ and $(it', val') \in S_k$. In either case the sequence of k operations and answers is incorrect. This completes case 2.

Remaining cases: Suppose the k th operation is one of the following: *insert*, *makeset*, *delete* or *merge*. None of these operations returns a value. Since we are assuming the sequences are partially correct it follows that the sequence of k operations

and answers is correct iff the sequence of $k - 1$ operations and answers is correct. It is not hard to show for each of these operations that $treeedges_k$ is an MST of G_k iff $treeedges_{k-1}$ is an MST of G_{k-1} . Thus the induction holds for each of these cases.

We shall do one example case. Suppose the k th operation is $delete(it)$. Note $treeedges_{k-1}$ and $treeedges_k$ are identical. Also, the sets $checkedges_{k-1}$ and $checkedges_k$ are nearly identical. $checkedges_k$ has the edge $(whereins_{k-1}(it), R)$ replaced with the following edge $(whereins_{k-1}(it), curroot_{k-1}(S))$. The weight of the edge, $(it, value(it))$, is unchanged. It is clear from the generalized MST definition that $treeedges_k$ is an MST of G_k iff $treeedges_{k-1}$ is an MST of G_{k-1} . This completes the cases and completes the proof. ■

Before giving our time complexity analysis we state a simple assumption. For our algorithm, we assume we have enough memory to maintain two arrays: the first is indexed by the item numbers of the pairs encountered, and the second is indexed by the set names of all of the sets encountered. (Note, we make no assumption like this about the item values, just the item numbers.) These arrays allow us in constant time to find the sets and pairs referenced by some operation. The above assumption is satisfied by all the applications of PQs and MPQs we have seen; however, if necessary we can “compress” the name space. Suppose the set names and item numbers are bounded by a polynomial in the length of the sequence that we are checking. We can in linear time use radix sort to efficiently “compress” the set names and item numbers used, so that the maximum size of a set name and item number is only linear in the length of the sequence being checked. The running times of all of our algorithms are expressed as a function of the number of operations being considered.

We now discuss the implementation of the Procedure *CheckMPQ*. First consider the operations $min(S)$ and $deletemin(S)$. The algorithm must check that a supposed answer to one of these operations, say (it, val) , was actually in the set S . These checks can be postponed to the end of the main loop and then a simple Depth First Search (DFS) procedure can check ancestor relationships in the tree given by $treeedges$. This allows all of these checks to be performed together in linear time. Now, consider the $delete$ operations. Our algorithm requires the name of the set from which the element was deleted. We can also postpone all of the processing of the $delete$ operations until the final stage. Then, the linear-time off-line union/find algorithm [40] can be used to process these operations. Since $delete$ operations only

create check edges, and check edges are only used during the final stage, the behavior of the algorithm is not altered by delaying this processing. Finally, Dixon et al. [31] show how the correctness of an MST as we generalized it can be checked in linear time.

Thus we have the following lemma:

Lemma 2.3.3: *Let O be a sequence of N MPQ operations, and let A be a sequence of supposed answers to these operations. The correctness of the answers in A can be determined in $O(N)$ time.*

2.3.2 A Practical Variant

We now present a theoretically less efficient version of our checker, but one which performs well in practice. First we will discuss the notion of computing functions on paths in trees [79]. Consider the following operations which allow a forest of trees to be incrementally constructed while query operations are performed on them:

1. *create*(v): Create a new tree containing only the node v .
2. *link*($v, w, weight$): Combine the trees with roots v and w into a single tree by making v the parent of w . Assign the edge (v, w) label *weight*.
3. *eval*(v): Find the root of the tree containing v , say r , and return the maximum of all labels on the path from r to v . If $v = r$ then *eval* returns $-\infty$.

Tarjan shows how to perform n *create* operations and m *eval* and *link* operations in $O((m+n)A(m+n, n))$ time, where $A(m+n, n)$ is the inverse of Ackermann's function. However, in our program, we use a simpler implementation using the basic technique of path compression. Although the worst case asymptotic behavior for our implementation is $O((n+m)\log(n+m))$, the algorithm performs better than this in practice. As a byproduct of either implementation, after an *eval*(v) operation is performed, the root r of the tree that v is contained in is also computed.

Based upon the *makeset*, *merge*, *min*, and *deletemin* operations, our algorithm constructs a forest of weighted trees using the operations *create* and *link*. Each *min* and *deletemin* operation creates an edge which is given the label of the supposed answer of that operation, and each *merge* operation creates two edges which are by

default given the label $-\infty$. There is a one to one correspondence between the roots of the trees and the sets which is given by the function *curroot*. Whenever a pair (it, val) is inserted into a set S , it is associated with the current root for S , and this association is given by the function *whereins*. Finally, the following invariant is maintained throughout the course of the algorithm. Let (it, val) be a pair currently in some set S . Let E be the labels of the edges of finite weight on the path from *whereins*(it) to *curroot*(S). Then the labels in E correspond precisely to the supposed answers to all of the *min* and *deletemin* operations acting on the sets of which (it, val) was a member. If (it, val) is smaller than any of the labels in E , then an error must have occurred earlier in the sequence since (it, val) is smaller than some pair which was reported as an answer. Thus, we use *eval* operations to determine if this possibility ever occurs. We now present the algorithm in full detail.

Procedure *CheckMPQ*(O, A)*:

Let \mathbf{S} be the set of all set names appearing in O .
 Let IT be the set of all item numbers appearing in O .
 Let \mathbf{R} be the universe of item values.
 Let Γ be a set of graph nodes.
 Let *curroot* be a function from \mathbf{S} to $\Gamma \cup \{nil, destroyed\}$.
 Let *whereins* be a function from IT to $\Gamma \cup \{nil, done\}$.
 Let *value* be a function from IT to \mathbf{R} .
 For each $S \in \mathbf{S}$ initialize *curroot*(S) to be *nil*.
 For each $it \in IT$ initialize *whereins*(it) to be *nil*.

The input is a sequence of N operations and supposed answers: O and A .
 If there is an error then routine *error* is called otherwise the answers are correct and the routine *ok* is called.
 For each i between 1 and N execute the appropriate case depending on i th operation:

Main loop:

makeset(S):

IF (*curroot*(S) \neq *nil*) THEN *error*.
 Let r be an unused node from Γ .
 Execute *create*(r), and set *curroot*(S) = r .

insert((it, val), S):

IF (*curroot*(S) = *nil*) OR (*curroot*(S) = *destroyed*) OR (*whereins*(it) \neq *nil*) THEN *error*.
 Set *whereins*(it) = *curroot*(S), and *value*(it) = val .

delete(it):

IF (*whereins(it) = nil*) OR (*whereins(it) = done*) THEN *error*.
 Set *tmp = eval(whereins(it))*.
 IF (*(it, value(it)) < tmp*) THEN *error*.
 Set *whereins(it) = done*.

min(S):

Let (*it, val*) be the supposed answer.
 IF (*whereins(it) = nil*) OR (*whereins(it) = done*) THEN *error*.
 IF (*value(it) ≠ val*) THEN *error*.
 Set *tmp = eval(whereins(it))*, and let *r* be the root of the tree containing *whereins(it)*.
 IF (*curroot(S) ≠ r*) THEN *error*.
 IF (*(it, val) < tmp*) THEN *error*.
 Let *r'* be an unused node from *Γ*.
 Execute *create(r')*, and perform *link(r', r, (it, val))*.
 Finally, set *curroot(S) = r'*.

deletemin(S):

Let (*it, val*) be the supposed answer.
 IF (*whereins(it) = nil*) OR (*whereins(it) = done*) THEN *error*.
 IF (*value(it) ≠ val*) THEN *error*.
 Set *tmp = eval(whereins(it))*, and let *r* be the root of the tree containing *whereins(it)*.
 IF (*curroot(S) ≠ r*) THEN *error*.
 IF (*(it, val) < tmp*) THEN *error*.
 Let *r'* be an unused node from *Γ*.
 Execute *create(r')*, and perform *link(r', r, (it, val))*.
 Set *curroot(S) = r'*.
 Set *whereins(it) = done*.

merge(S₁, S₂, S):

IF (*curroot(S₁) = nil*) OR (*curroot(S₁) = destroyed*) THEN *error*.
 IF (*curroot(S₂) = nil*) OR (*curroot(S₂) = destroyed*) THEN *error*.
 IF (*curroot(S) ≠ nil*) THEN *error*.
 Let *r* be an unused node from *Γ*.
 Perform *link(r, curroot(S₁), (θ, -∞))* and *link(r, curroot(S₂), (θ, -∞))*.
 Set *curroot(S)* to be *r*.
 Set *curroot(S₁)* and *curroot(S₂)* equal to *destroyed*.

Final stage:

For every *it* such that *whereins(it) ≠ nil* and *whereins(it) ≠ done*,
 check that *eval(whereins(it)) ≤ (it, value(it))*.

IF all these checks succeed THEN *ok* ELSE *error*.

Lemma 2.3.4: *Algorithm CheckMPQ* is correct.*

Proof: We note that this program can be viewed as an incremental version of the checker given previously, and the proof of correctness is similar to the one given before as well. ■

The time required by procedure *CheckMPQ** is dominated by the time it takes to perform the *create*, *link* and *eval* operations. As mentioned before this can be done in $O(nA(n, n))$ time, where n is the length of the sequence.

Lemma 2.3.5: *Let O be a sequence of N MPQ operations, and let A be a sequence of supposed answers to these operations. Then *CheckMPQ** checks the correctness of the answers in A in $O(NA(N, N))$ time.*

As the algorithm is currently described, many errors can be detected during the execution of the Main Loop. However, some errors in the sequence of supposed answers will not be detected until the execution of the Final Stage. If less latency in the detection of errors is desired, one could periodically execute the operations in the Final Stage, and then resume execution in the Main Loop. For example, if it was anticipated that 1,000,000 MPQ operations were going to be performed in total, then after every 50,000 operations the *eval* computations and checks could be performed on all of the pairs which are active. If all of these checks succeed, then no errors could have occurred up until that point. Provided that the total number of pairs which we need to check during any execution of the code in the Final Stage doesn't grow too large (in our example, say less than 25,000), then the total computational cost wont grow significantly. This is because after executing the steps in the Final Stage each tree in the forest can be collapsed into a single node, thus speeding up future *eval* operations. This compression of trees into nodes also has the side effect of providing superior space utilization. Other variations on the strategy to reduce latency are possible.

2.4 Timing Results

In this section we describe how we performed the timing experiments comparing the speed of the PQ and MPQ implementations to that of the practical version of the MPQ checker. All of our timing experiments were performed on a SUN-ELC with 32 megabytes of main memory. All programs were compiled with the Sun Pascal Compiler, with the highest optimization level (-O4) enabled. Since the compiler initializes all static arrays to zero, explicit initialization was not required in our checker or in the driver to the MPQ implementations. Also, in both the checker and the MPQ implementations, we replaced all calls to the Pascal allocation function “new()” with our own allocation routine, which preallocated all of the required records outside of the main timing loop.

We tested the checker under a variety of parameter values, such as number of priority queues, and relative probabilities of performing *insert*, *deletemin* and *merge* operations. For a fixed set of parameters, we generated five random sequences and averaged the timings for all of them. The variance between runs was small, despite the fact that each run had a different randomly generated sequence. We have not performed extensive timing for *delete* and *min* operations, though we anticipate the results would be similar. All data sets were timed for all of the different PQ and MPQ implementations. In our tables, PHp is pairing heap [35], SHp is skew heap [73], BHp is binomial queue [82], LHp is leftist tree [57], and FHp is Fibonacci heap [36].

Our first set of test cases consisted of a sequence of operations on only one priority queue. Here, N *insert* operations were followed by N *deletemin* operations, for several different values of N . See Table 2.1 for the results. The data for Check is the (average) time in milliseconds which our checker required to run. The data for PHp, SHp, BHp, LHp, and FHp is how many times longer the respective implementation took to run on the same data sets.

Our second set of test cases again consisted of a sequence of operations on only one priority queue. In the first phase, N *insert* operations were performed. The second phase consisted of $2N$ operations, where each operation was an *insert* with 50% probability and a *deletemin* with 50% probability. During the third phase *deletemin* operations were performed until the priority queue was empty (thus one would expect the final phase to consist of approximately N *deletemin* operations.)

N	Check	PHp	SHp	BHp	LHp	FHp
5000	77	2.35	3.27	3.35	5.00	6.27
10000	163	2.55	3.26	3.43	4.94	6.24
20000	310	2.82	3.74	3.99	5.80	7.10
50000	807	3.09	3.98	4.26	6.20	7.49
100000	1663	3.21	4.12	4.49	6.52	7.78

Table 2.1: 1 Priority Queue, N *Insert* followed by N *Deletemin* Operations

N	Check	PHp	SHp	BHp	LHp	FHp
5000	217	1.63	2.26	2.23	3.38	4.29
10000	437	1.75	2.40	2.43	3.71	4.58
20000	907	1.80	2.46	2.53	3.82	4.74
50000	2366	1.92	2.54	2.67	4.07	4.82

Table 2.2: 1 Priority Queue, N *Insert*, 2N *Insert* and *Deletemin*, N *Deletemin*

See Table 2.2 for our results.

For our third set of test cases we had the parameters N , P , T , and E . N is the number of operations, P is the number of priority queues, T is the target size of a priority queue, and E is the expected number of *merge* operations to be performed. Since we fix the number of priority queues active at any time, after each *merge* operation we always create a new priority queue. Each operation has an E/N chance of being a *merge* operation. If it is a *merge*, then the first two arguments to the merge are chosen randomly, where the probability of any particular priority queue Q being chosen is proportional to $1/size(Q)$, where $size(Q)$ is the number of pairs currently in Q . Intuitively, a small priority queue is more likely to be an argument to a *merge* than a large priority queue. If the operation is not randomly chosen to be a *merge*, then we randomly select a priority queue, with the probability that a particular priority queue Q is chosen is proportional to $|T - size(Q)|$. Intuitively, a priority queue which has either significantly more or fewer elements than the target size is more likely to be chosen than a priority queue which is close to the target size. After the priority queue is chosen, we perform either an *insert* or a *deletemin* operation depending on whether the priority queue is smaller than or larger than the target size. See Tables 2.3 and 2.4 for our results. Our probability distributions bias

PQs	Trgt	Chk	PHp	SHp	BHp	LHp	FHp
5	2000	1973	1.79	2.48	2.41	3.52	4.68
5	4000	2047	1.88	2.52	2.51	3.68	4.74
5	6000	2000	1.92	2.59	2.58	3.81	4.85
5	8000	2017	1.97	2.59	2.68	3.92	4.77
10	1000	1930	1.75	2.40	2.37	3.44	4.52
10	2000	2017	1.87	2.50	2.50	3.63	4.65
10	3000	2037	1.91	2.56	2.59	3.78	4.71
10	4000	2027	1.94	2.58	2.67	3.87	4.69
10	5000	2007	1.97	2.55	2.72	3.93	4.63
20	1000	1947	1.81	2.44	2.44	3.56	4.52
20	2000	1956	1.91	2.55	2.64	3.82	4.61
20	3000	1920	1.92	2.51	2.75	3.91	4.49

Table 2.3: 200,000 Operations, 200 Expected Merges

PQs	Trgt	Chk	PHp	SHp	BHp	LHp	FHp
50	1000	4020	1.87	2.48	2.51	3.64	4.59
50	2000	4170	1.88	2.45	2.59	3.72	4.40
100	1000	4017	1.81	2.42	2.57	3.69	4.30

Table 2.4: 400,000 Operations, 400 Expected Merges

a *merge* operation to operate on small priority queues, *insert* operations to act on small priority queues, and *deletemin* operations to operate on larger priority queues. This has the effect of keeping the priority-queue sizes relatively balanced and near the target size. Randomly choosing the priority queues for operations without regard to their size tends to result in a few priority queues containing the vast majority of the elements, and the rest having relatively few elements.

2.5 Equivalence Between Checking MSTs and Sequences of MPQ Operations

In the previous sections, we showed that it is possible in linear time to reduce an instance of checking the correctness of a sequence of mergeable priority queue operations to an instance of checking that a tree is a minimum spanning tree of a graph.

In this section, we show that the converse is true. That is, it is possible in linear time to reduce an instance of checking that a tree is a minimum spanning tree of a graph to an instance of checking the correctness of a sequence of mergeable priority queue operations.

Let T be a spanning tree of a graph G . For any edge e in G , let $w(e)$ be the weight of e . First, pick an arbitrary node r to be the root of T . For simplicity, we assume that each node in T has exactly two children. It is easy to modify the algorithm when this is not the case. Replace each non-tree edge (x, y) in G such that x is neither an ancestor nor a descendant of y with the two edges (w, x) and (w, y) , where w is the lowest common ancestor of x and y when T is rooted at r . Both (w, x) and (w, y) are given the weight $w(x, y)$. Assign each node x in T a unique positive integer identifier $l(x)$. Also, assign each edge e in G a unique positive integer identifier $l(e)$. By notational convention, if we have an edge $e = (x, y)$, then sometimes we write $l(x, y)$ instead of $l(e)$. Initialize Seq to be a sequence of MPQ operations which is initially empty. The sequence contained in Seq when the following procedure terminates will be the MPQ sequence to which the MST check problem for T is reduced.

Perform a depth first search on T starting from r , and visit all of the nodes in T in a post-order fashion. When we visit a node w which is a leaf, we append the operation $create(l(w))$ onto the end of Seq . In addition, while visiting a leaf node w , for each non-tree edge $e = (v, w)$, we append the operation $insert((l(e), w(e)), l(w))$ onto Seq .

Now assume we are currently visiting the internal node w with children x and y . Since we are traversing T in a post-order fashion, we have already recursed on both x and y . Append the following operations onto the end of Seq :

$$\begin{aligned} &insert((l(w, x), w(w, x)), l(x)) \\ &dmin(l(x)) = (l(w, x), w(w, x)) \\ &insert((l(w, y), w(w, y)), l(y)) \\ &dmin(l(y)) = (l(w, y), w(w, y)) \\ &merge(l(x), l(y), l(w)). \end{aligned}$$

For each non-tree edge $e = (w, z)$ such that z is a descendant of w , append the operation $delete(l(e))$ onto Seq . Finally, for each non-tree edge $e = (v, w)$ such that v is an ancestor of w , append the operation $insert((l(e), w(e)), l(w))$ onto Seq .

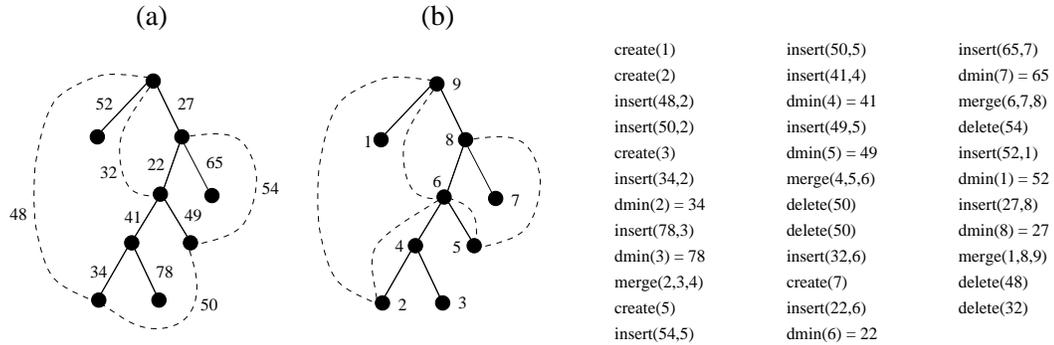


Figure 2.3: Reduction of MST check to MPQ operation sequence check.

See Figure 2.3 for an example. We wish to check that the solid edges in (a) form a minimum spanning tree of the graph in (a). The vertices of the tree are labeled with unique positive integers according to a post-order traversal of the tree (b). Also, notice that the edge with weight 50 in (a) has been split into two separate edges in (b). For clarity in the figure, the numbering on the edges that is generated by the procedure is omitted, and in addition we omit the item numbers of the elements in the output sequence. Thus, the operation *delete* is now given the weight of the element being deleted instead of the item number. The operation sequence generated is presented in three columns: the operations in the first column come before the operations in the second column, which come before the operations in the third column. The sequence of MPQ operations is correct, and so by the next theorem we see that the MST must also be correct.

Lemma 2.5.1: *Let T be a spanning tree of G . Then T is an MST of G if and only if the sequence generated by the above procedure (with the given answers) is correct.*

Proof: First we will prove that if T is an MST of G , then the sequence generated is correct. By way of contradiction, assume that the sequence generated is incorrect. So consider the first operation in the sequence of the form $dmin(S) = a$ such that a is incorrect. This operation must have been produced when visiting some interval node w of T . Let x and y be the children of w . After recursing on x and y , the following operations will be output (followed by some *delete* and then *insert* operations):

$$\begin{aligned}
& \text{insert}((l(w, x), w(w, x)), l(x)) \\
& \text{dmin}(l(x)) = (l(w, x), w(w, x)) \\
& \text{insert}((l(w, y), w(w, y)), l(y)) \\
& \text{dmin}(l(y)) = (l(w, y), w(w, y)) \\
& \text{merge}(l(x), l(y), l(w)).
\end{aligned}$$

It must be the case that either $S = l(x)$ or $S = l(y)$. Since both cases are essentially the same, here we will only consider the case when $S = l(x)$. So we see that $a = (l(w, x), w(w, x))$. Since a is not the correct answer to $\text{dmin}(S)$, and a was in $l(x)$, there must be a pair b in $l(x)$ such that $b < a$ holds. Call all *insert* operations in the sequence that are immediately followed by a *dmin* operation Type 1 inserts, and all the other *insert* operations Type 2 inserts. Since $\text{dmin}(S) = a$ is the first incorrect operation, it must be the case that b was originally inserted by a Type 2 insert. Let the non-tree edge associated with b be (v, z) (with v the ancestor of z). Clearly, a *delete* operation for b will be issued when node v is visited. We can prove that v is an ancestor of w (or is w), and that z is a descendant of x (or is x). However, this violates the definition of a minimum spanning tree, since we have a non-tree edge (v, z) whose weight is less than the weight of an edge along the path from v to z (in this case the edge (w, x)).

Now we will prove that if the sequence is correct, then T must be an MST of G . By way of contradiction, assume that T is not an MST of G . So consider the first tree edge traversed, call it (w, x) , such that there is a non-tree edge (v, z) with $w(v, z) < w(w, x)$, and (w, x) is along the path from v to z . Assume that w is the parent of x , and that v is an ancestor of z . We can prove that a pair with the weight of the edge (v, z) must be contained in $l(w)$ as the processing on w begins. But since $w(v, z) < w(w, x)$, the operation $\text{dmin}(l(x)) = (l(w, x), w(w, x))$ is incorrect, which is a contradiction. ■

Note that the existence of a linear-time algorithm for checking the MST property of a tree implies a trivial solution for this problem. Let Seq_1 be a sequence that is correct, and let Seq_2 be a sequence that is incorrect. (For example, $Seq_1 = (\text{insert}((1, 10), 1), \text{insert}((2, 20), 1), \text{dmin}(1) = (1, 10))$ and $Seq_2 = (\text{insert}((1, 10), 1), \text{insert}((2, 20), 1), \text{dmin}(1) = (2, 20))$ will suffice.) Given a tree T and a graph G , we could use the linear-time algorithm to determine if T is an MST of G . If it is, then we would output Seq_1 as the corresponding MPQ check

problem, and if it is not, then we would output Seq_2 as the corresponding MPQ check problem.

One can ask why we presented the first reduction when there is a trivial reduction using the linear-time MST checker. The reason is that the first reduction shows there is a close relationship between checking sequences of mergeable priority queues operations and checking minimum spanning trees. We feel that the first reduction we presented gives added insight into both of these problems.

2.6 Checking Splittable Priority Queues

In this section we present an algorithm to check the correctness of a sequence of SPQ operations. Let $O = (o_1, o_2, \dots, o_n)$ be a sequence of SPQ operations, and let $A = (a_1, a_2, \dots, a_n)$ be the supposed answers to the operations in O . In this section we assume that the operation $delete(it)$ returns the name of the set from which the pair with item number it was deleted. The semantics of $deletemin(S)$ are unchanged.

For clarity, in the procedure we present we omit the following checks:

1. That set names and item numbers are never reused.
2. That when the operation $delete(it)$ is performed, that a pair with item number it is present in some set (but assuming that it is in some set, our algorithm checks that it is in the set given by the supposed answer).
3. That the pair given as the supposed answer to a $min(S)$ or $deletemin(S)$ query must be in some set (but assuming that the pair is in some set, our algorithm tests that the pair is actually in the set S).

These checks would of course have to be present in an actual implementation, and would be similar to the ones given in *CheckMPQ*. Any sequence of operations and supposed answers which satisfies these checks is said to be partially correct.

We now give a high level description of how *CheckSPQ* works. Again, we process the operations in O one by one. Based on the *makeset*, *min*, *deletemin*, *delete*, and *split* operations we construct two separate forests of weighted trees. The trees from the first forest are thought of as having positive constraints, and the trees from the second forest are thought of as having negative constraints. When a pair is reported as a supposed answer, a positive constraint is created which checks that no other pair in the set had a smaller value. When the operation $split(S, val, S_1, S_2)$ is

processed, a positive constraint is created enforcing that no pair smaller than (∞, val) is placed in S_2 . Also, a negative constraint is created enforcing that no pair larger than (∞, val) is placed in S_1 . For each pair in O , we create two check edges, one for the forest of positive constraints, and one for the forest of negative constraints. As a last step, all of the trees in the forest of positive constraints are hooked to one root, and all of the trees in the forest of negative constraints are hooked to another root. As before, A contains the correct answers to O if and only if the final two trees are MSTs of their respective graphs.

At any point during the execution of the algorithm, we define $whichset(it)$ to be the name of the set which currently contains the pair with item number it . If no pair with item number it is contained in any set, then $whichset(it) = nil$.

See Figure 2.4 which uses the same style for abbreviations and annotations as Figure 2.2. In this figure, *split* is sometimes abbreviated as just the letter “s”. As in the earlier figure, we do not include the item numbers of the elements. Some of the numbers in the figure are marked with “*”, and this affects how the number is treated in a comparison. We define $x < x^*$ to be true, but the relative order of x^* and y is the same as for x and y if $x \neq y$ holds. Unlabeled edges are assumed to have a default weight of $(0, -\infty)$.

Procedure *CheckSPQ*:

Let \mathbf{S} be the set of all set names appearing in O .

Let IT be the set of all item numbers appearing in O .

Let Γ be a set of graph nodes.

Let $treeedges^+$ and $treeedges^-$ be subsets of $\Gamma \times \Gamma$ which are initially empty.

Let $checkedges^+$ and $checkedges^-$ be subsets of $\Gamma \times \Gamma$ which are initially empty.

Let $BeginSets$ be a subset of \mathbf{S} which is initially empty.

Let $weight$ be a function from $checkedges \cup treeedges$ to $IT \times \mathfrak{R}$.

Let $curnode^+$ and $curnode^-$ be functions from IT to $\Gamma \cup \{nil, destroyed\}$.

Let $whereins^+$ and $whereins^-$ be functions from IT to $\Gamma \cup \{nil, done\}$.

Let $value$ be a function from IT to \mathfrak{R} .

For each $S \in \mathbf{S}$, initialize $curnode^+(S)$ and $curnode^-(S)$ to nil .

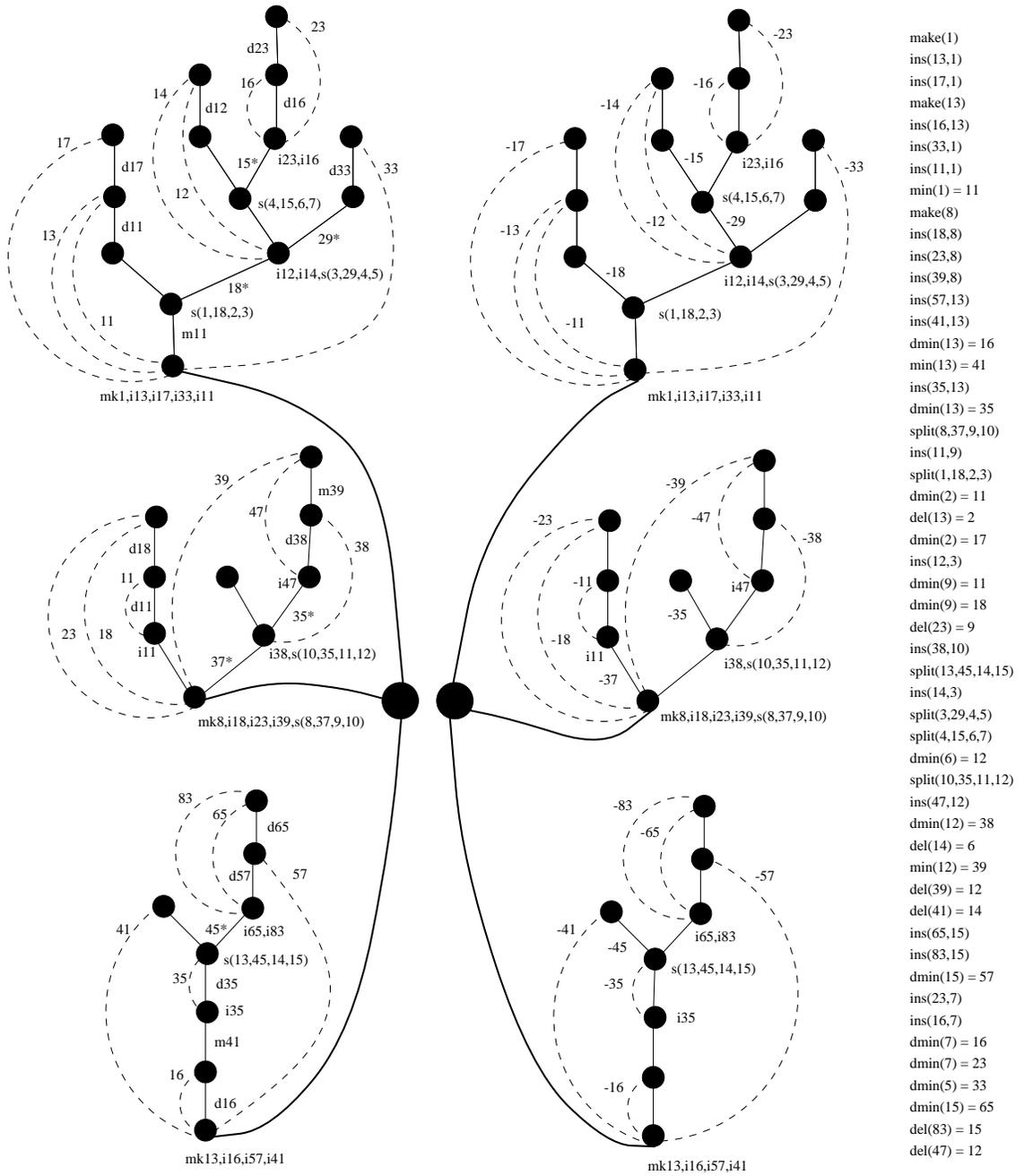
For each $it \in IT$, initialize $whereins^+(it)$ and $whereins^-(it)$ to nil .

For each i between 1 and N execute the appropriate case depending on the i th operation:

Main loop:

makeset(S):

Figure 2.4: SPQ example.



Let r^+ and r^- be two unused nodes from Γ , and set
 $currnode^+(S) = r^+$ and $currnode^-(S) = r^-$.
 Add S to *BeginSets*.

insert((it , val), S):

Set $whereins^+(it) = currnode^+(S)$, and $whereins^-(it) = currnode^-(S)$.
 Set $value(it) = val$.

delete(it):

Let S be the set which supposedly contains the pair with item number it .
 Let $u^+ = whereins^+(it)$ and $v^+ = currnode^+(S)$.
 Add the edge (u^+, v^+) to the set *checkedges*⁺,
 and set $weight(u^+, v^+) = (it, value(it))$.
 Let $u^- = whereins^-(it)$ and $v^- = currnode^-(S)$.
 Add the edge (u^-, v^-) to the set *checkedges*⁻,
 and set $weight(u^-, v^-) = (-it, -value(it))$.
 Set $whereins^+(it) = done$.

min(S):

Let (it , val) be the supposed answer.
 Add ($currnode^+(S)$, $whereins^+(it)$) to the set *checkedges*⁺,
 and set $weight(currnode^+(S), whereins^+(it)) = (it, val)$.
 Add ($currnode^-(S)$, $whereins^-(it)$) to the set *checkedges*⁻,
 and set $weight(currnode^-(S), whereins^-(it)) = (-it, -val)$.
 Let r^+ and r^- be two unused nodes from Γ .
 Add (r^+ , $currnode^+(S)$) to the set *treeedges*⁺,
 and set $weight(r^+, currnode^+(S)) = (it, val)$.
 Add (r^- , $currnode^-(S)$) to the set *treeedges*⁻,
 and set $weight(r^-, currnode^-(S)) = (0, -\infty)$.
 Set $currnode^+(S) = r^+$, and $currnode^-(S) = r^-$.

deletemin(S):

Let (it , val) be the supposed answer.
 Let r^+ and r^- be two unused nodes from Γ .
 Add (r^+ , $currnode^+(S)$) to the set *treeedges*⁺,
 and set $weight(r^+, currnode^+(S)) = (it, val)$.
 Add (r^- , $currnode^-(S)$) to the set *treeedges*⁻,
 and set $weight(r^-, currnode^-(S)) = (0, -\infty)$.
 Set $currnode^+(S) = r^+$, and $currnode^-(S) = r^-$.
 Let $v^+ = whereins^+(it)$, and $v^- = whereins^-(it)$.
 Add the edge (v^+, r^+) to the set *checkedges*⁺,
 and set $weight(v^+, r^+) = (it, val)$.
 Add the edge (v^-, r^-) to the set *checkedges*⁻,

and set $weight(v^-, r^-) = (-it, -val)$.
 Set $whereins^+(it) = done$.

split(S, val, S_1, S_2):

Let l^+, l^-, r^+ , and r^- be four unused nodes from Γ .

Let $c^+ = curnode^+(S)$, and $c^- = curnode^-(S)$.

Add the edge (l^+, c^+) to the set $treeedges^+$, and set $weight(l^+, c^+) = (0, -\infty)$.

Add the edge (r^+, c^+) to the set $treeedges^+$, and set $weight(r^+, c^+) = (\infty, val)$.

Add the edge (l^-, c^-) to the set $treeedges^-$, and set $weight(l^-, c^-) = (-\infty, -val)$.

Add the edge (r^-, c^-) to the set $treeedges^-$, and set $weight(r^-, c^-) = (0, -\infty)$.

Set $curnode^+(S_1) = l^+$, and $curnode^-(S_1) = l^-$.

Set $curnode^+(S_2) = r^+$, and $curnode^-(S_2) = r^-$.

Set $curnode^+(S) = destroyed$ and $curnode^-(S) = destroyed$.

Final stage:

For every pair (it, val) such that $whereins^+(it) \neq done$ add the edge

$(whereins^+(it), curnode^+(whichset(it)))$ to the set $checkedges^+$, and assign the edge weight (it, val) . Also, add the edge $(whereins^-(it), curnode^-(whichset(it)))$ to the set $checkedges^-$, and assign the edge weight $(-it, -val)$.

For every edge (u, v) in $checkedges^+$, verify that u and v are in the same tree

in the forest defined by $treeedges^+$, and also that u is an ancestor of v (or vice versa).

(* these are called the “couldbe” checks *)

Let R^+ and R^- be two unused nodes from Γ .

For each set $S \in BeginSets$, add the edge

$(R^+, curnode^+(S))$ to $treeedges^+$ and $(R^-, curnode^-(S))$

to $treeedges^-$ and give both edges weight $(0, -\infty)$.

Let Γ' refer to the nodes actually used by the algorithm.

Let G^+ be the graph formed by the edges in $treeedges^+$ and $checkedges^+$.

Let G^- be the graph formed by the edges in $treeedges^-$ and $checkedges^-$.

IF $treeedges^+$ is an MST of G^+ AND

$treeedges^-$ is an MST of G^- THEN *ok*

ELSE *error*.

The following lemma proves to be useful.

Lemma 2.6.1: Assume we have processed the first k operations. Let (it, val) be a pair, and assume that the following three conditions hold:

1. $whereins^+(it)$ is a set name.
2. $value(it) = val$.

3. The first k operations with supposed answers are correct.

Then $whichset(it) = S$ if and only if the follow three conditions hold:

- 4 $whereins^+(it)$ is an ancestor of $curnode^+(S)$.
- 5 (it, val) is bigger than or equal to the weights of all the edges on the path from $whereins^+(it)$ to $curnode^+(S)$ that were created by a *split* operation.
- 6 $(-it, -val)$ is bigger than or equal to the weights of all the edges on the path from $whereins^-(it)$ to $curnode^-(S)$ that were created by a *split* operation.

Proof: We will prove this by induction on k . So assume that the theorem is true for $k = i - 1$. We must show that the theorem is true for $k = i$.

Consider the case when the i th operation is of the form $insert((it', val'), S')$. There are two possibilities. The first is that $(it', val') = (it, val)$. It is clear that condition 4 will only hold for the set S' . Since conditions 5 and 6 obviously hold for the set S' , we are done with this possibility. Next, it is possible that $(it', val') \neq (it, val)$. Clearly, conditions 1, 2, and 3 must hold after the $(i - 1)$ th operation if they hold after the i th operation. So by induction, conditions 4, 5, and 6 only hold for the set S which contains (it, val) after the the $(i - 1)$ th operation is processed. Whether or not $S = S'$, condition 4 will only hold for the set S after the i th operation. Since the i th operation is not a *split*, if conditions 5 and 6 held after the $(i - 1)$ th operation for set S , they must hold after the i th operation as well. So we are done with the case that the i th operation is an *insert*.

Consider the case when the i th operation is of the form $split(S', val', S'_1, S'_2)$. Consider the possibility that $S \neq S'_1$ and $S \neq S'_2$. By induction, conditions 4, 5, and 6 held for the set S after the $(i - 1)$ th operation, so clearly conditions 4, 5, and 6 will hold for the set S after the i th operation is processed. Since neither S'_1 nor S'_2 contain (it, val) after the i th operation, we see that S' could not have contained (it, val) after the $(i - 1)$ th operation. Since S' did not contain (it, val) after the $(i - 1)$ th operation, by induction we see that one of the conditions 4, 5, or 6 must not have held for S' . But then this same condition will not hold for either of S'_1 or S'_2 after the i th operation. Similarly, consider any other set $S'' \neq S$ existing after the i th operation. Clearly, S'' can not contain (it, val) after the i th operation, and hence can not contain (it, val) after the $(i - 1)$ th operation. By induction, one of the conditions 4, 5, or 6 must have

been violated for S'' after the $(i - 1)$ th operation. Clearly, this same condition must be violated after the i th operation as well.

Now consider the possibility that $S'_1 = S$. Consider any set S'' that exists after the i th operation such that $S'' \neq S'_1$ and $S'' \neq S'_2$. Clearly, S'' can not contain (it, val) after the i th operation, and hence can not contain (it, val) after the $(i - 1)$ th operation. By induction, one of the conditions 4, 5, or 6 must have been violated for S'' after the $(i - 1)$ th operation. Clearly, this same condition must be violated after the i th operation as well. Now consider the set S'_2 . While conditions 4 and 6 will hold for S'_2 after the i th operation, condition 5 will be violated. This is because *split* creates a new tree edge $(curnode_{i-1}^+(S'), curnode_i^+(S'_2))$ and gives this edge weight (∞, val') . We know that $(it, val) < (\infty, val')$ because (it, val) belongs in set S'_1 and not S'_2 , and hence condition 5 is violated. Finally, we must show that conditions 4, 5, and 6 hold for S'_1 after the i th operation. Note that S' must have contained (it, val) after the $(i - 1)$ th operation is processed. First, we see that condition 4 clearly holds for S'_1 , since by induction it must have held for S' after the $(i - 1)$ th operation. Also by induction, we see that the only way that conditions 5 or 6 can be violated for S'_1 is if any of the new edges created by the *split* operation being performed causes the violation. But, $(it, val) < (\infty, val')$, since (it, val) is in set S'_1 . So we see that $(-it, -val) \geq (-\infty, -val')$ (item numbers are finite, so even if $val = val'$, we have that $-it > -\infty$), and hence condition 6 can not be violated. Also, $(it, val) > (0, -\infty)$, so condition 5 can not be violated.

The final possibility for the *split* operation is that $S'_2 = S$. The proof for this possibility is very similar to when $S'_1 = S$, and we do not present the details here.

The final cases are when the i th operation is a *makeset*, *min*, *delete*, or *deletemin* operation. Using similar techniques to the ones developed above, we can prove the theorem is true for these cases as well. ■

Lemma 2.6.2: *Algorithm CheckSPQ is correct.*

Proof: It is not difficult to show that CheckSPQ behaves correctly for sequences which are not partially correct. Thus, as before, we will assume that the input sequences are partially correct. This means that the algorithm will not stop in the main loop with a call to *error* but will always execute the final stage. Also, we assume that all of the “couldbe” checks succeed. It is clear that if any of the checks fail then the sequence is not correct.

A simple induction on the number of iterations of the main loop can be used to prove the loop invariants given next. If $whereins^+(it) = nil$ then item it has not been inserted yet. If $whereins^+(it) = done$ then item it has been inserted and deleted. Otherwise, item it has been inserted and not yet deleted. If $curnode^+(S) = nil$ then set S has not been created by *makeset* (or *split*) yet. If $curnode^+(S) = destroyed$ then set S has been created by *makeset* (or *split*) and destroyed by a *split* operation. Otherwise, S has been created but not yet destroyed. Again, let us call such a set *active*.

Let us use Γ' to refer to nodes which have actually been used by the algorithm at some point during execution. Let Γ^+ be the nodes in Γ' which are incident to edges in $treeedges^+$, and let Γ^- be the nodes in Γ' which are incident to edges in $treeedges^-$. It is clear that Γ^+ and Γ^- partition the nodes of Γ' . The following invariants can also be easily proven. The edge set $treeedges^+$ and the vertex set Γ^+ form a forest. Further, there is a one-to-one correspondence between the root nodes of all of the trees in the forest and the *makeset* operations processed up to that point. Also, there is a one-to-one correspondence between the leaves of the trees in the forest and the currently active sets. This correspondence is given by the function $curnode^+$. Analogous invariants are also true of Γ^- , $treeedges^-$, and $curnode^-$.

For our main proof we use a different induction. We assume that the behavior of CheckSPQ is correct for all sequences of length $k - 1$ and we attempt to show it is correct for all sequences of length k . The base case for the induction is trivially true.

Let $treeedges_{k-1}^+$ be the value of $treeedges^+$ when the algorithm completes execution after being given only the first $k - 1$ operations and answers as input. Similarly, we use the subscripts k and $k - 1$ on other sets and functions with the same meaning as above. Let G_k^+ refer to the graph with vertices Γ_k^+ and edges $treeedges_k^+ \cup checkedges_k^+$. Similarly, we define G_k^- .

Recall, we are assuming that the operations and answers are partially correct and thus the final phase is always executed. Also, recall that we assume that all the “couldbe” checks succeed. Since CheckSPQ behaves correctly on sequences of length $k - 1$ we know by examination of the final test performed for CheckSPQ that the first $k - 1$ operations and answers are correct iff $treeedges_{k-1}^+$ is an MST of the graph G_{k-1}^+ and that $treeedges_{k-1}^-$ is an MST of the graph G_{k-1}^- . To complete our proof we must

show that the first k operations and answers are correct iff $treeedges_k^+$ is an MST of G_k^+ and that $treeedges_k^-$ is an MST of G_k^- .

To show this we perform a case analysis for each possible k th operation.

Case 1: Suppose the k th operation is $min(S)$ and the supposed answer is (it, val) . Note that $treeedges_{k-1}^+$ and $treeedges_k^+$ are nearly identical. $treeedges_k^+$ has the additional edge $(curnode_k^+(S), curnode_{k-1}^+(S))$ which has weight (it, val) . Also, $treeedges_k^-$ has the additional edge $(curnode_k^-(S), curnode_{k-1}^-(S))$ which has weight $(0, -\infty)$.

Now consider the differences between the sets $checkedges_k^+(S)$ and $checkedges_{k-1}^+(S)$. Every edge $(u, curnode_{k-1}^+(S))$ in $checkedges_{k-1}^+(S)$ that was created in the final stage is replaced with the edge $(u, curnode_k^+(S))$ in $checkedges_k^+(S)$ (which will also be created in the final stage). Also, the edge $(whereins_{k-1}^+(it), curnode_{k-1}^+(S))$ will be in $checkedges_k^+$ and will have weight (it, val) . Now consider the differences between the sets $checkedges_k^-(S)$ and $checkedges_{k-1}^-(S)$. The changes will be very similar to that between sets $checkedges_k^+(S)$ and $checkedges_{k-1}^+(S)$, except that the edge $(whereins_k^-(it), curnode_{k-1}^-(S))$ in $checkedges_k^-$ will have weight $(-it, -val)$.

Suppose that $treeedges_k^+$ is an MST of G_k^+ and $treeedges_k^-$ is an MST of G_k^- . By the MST theorem above and the observations above we immediately conclude that $treeedges_{k-1}^+$ is an MST of G_{k-1}^+ and $treeedges_{k-1}^-$ is an MST of G_{k-1}^- . Thus the first $k-1$ operations and answers are correct. Recall that the k th operation is of the form $min(S)$, and has supposed answer (it, val) . We need to show that the supposed answer to the min is correct. Recall that we assumed all of the “couldbe” checks succeeded, so specifically we have that $whereins_{k-1}^+(it)$ is an ancestor of $curnode_{k-1}^+(S)$. This, in conjunction with the facts that $treeedges_k^+$ is an MST of G_k^+ , $treeedges_k^-$ is an MST of G_k^- , and the lemma proved above, we see that $whichset_k(it) = S$ and hence (it, val) is actually contained in the set S at time k . Note, that for every pair $(it', val') \in S$ at time k there is an edge $(curnode_k^+(S), whereins_k^+(it'))$ with weight (it', val') in G_k^+ . Note, the edge $(curnode_k^+(S), curnode_{k-1}^+(S))$ which has weight (it, val) is on the simple path from $curnode_k^+(S)$ to $whereins_k^+(it')$ in $treeedges_k^+$ for every pair $(it', val') \in S$ at time k . By the MST theorem, it follows that for every $(it', val') \in S$ at time k , $(it', val') \geq (it, val)$. By definition of the min operation we conclude that the supposed answer is correct.

Now suppose that either $treeedges_k^+$ is not an MST of G_k^+ , or $treeedges_k^-$

is not an MST of G_k^- . By the MST theorem it is not hard to show that either $treeedges_{k-1}^+$ is not an MST of G_{k-1}^+ , $treeedges_{k-1}^-$ is not an MST of G_{k-1}^- , or there exists an edge of the form $(curnode_k^+(S), whereins_k^+(it'))$ with weight (it', val') such that $(it', val') < (it, val)$ and $(it', val') \in S$ at time k . In any of the cases the sequence of k operations and answers is incorrect. This completes case 1.

The other cases are when the i th operation is a *makeset*, *split*, *insert delete*, or *deletemin* operation. Using similar techniques to the ones developed above, we can prove the theorem is true for these cases as well. ■

Now consider the implementation of *CheckSPQ*. Again we use linear-time MST checker as a subroutine [31]. Also, all of the “couldbe” checks in the final stage can be performed using any optimal least common ancestor preprocessing algorithm. The only difficulty is the first step in the final stage, where we need to determine the set that contains each undeleted pair. There are several possibilities. The simplest possibility is to assume that all of the priority queues are empty after the last operation is processed. If this is not the case, the original algorithm which uses the SPQ can be modified to *delete* all the pairs before terminating. Now the full sequence of operations can be checked. Alternatively, the algorithm which implements the splittable priority queues can be modified to output the set location of each pair which has not been deleted after the last operation has been processed. Finally, the program checker could determine which set each leftover pair is contained in during the final stage. This can be implemented by performing a second pass over the input sequence, processing only *split* operations, and also all of the *insert* operations which act on pairs which are leftover at the end of the input sequence. It is not difficult to show that a sequence of length m containing m_i *insert* operations and the rest *split* operations can be processed in $O(m_i \log m_i + m)$ time. So we see that if in the original sequence of n operations, at most $O(n/\log n)$ pairs remain after the last operation is processed, the splittable priority queue checker will run in $O(n)$ time.

Chapter 3

On-Line Certification of Mergeable and Splittable Priority Queues

In this chapter we consider checking sequences of mergeable priority queue and splittable priority queue operations in an on-line fashion. In order to obtain useful results we use the certification trail method.

It is important to understand clearly how the algorithms presented in this chapter differs from the algorithms *CheckMPQ* and *CheckSPQ* presented in the last chapter. Both *CheckMPQ* and *CheckSPQ* are off-line algorithms, meaning that they are given the entire sequences of operations and supposed answers at the start of execution, and then they determine if the supposed answers are correct. The algorithm *CheckMPQ** is presented the operations and the supposed answers one by one, and performs useful processing before the next operation and supposed answer are received. Although many errors can be detected soon after they occur, many others are not detected until many operations later. Depending upon the application, this latency in detection can lead to the propagation of errors, which can have undesirable consequences.

In this section we present on-line certifiers for both mergeable priority queues and splittable priority queues. At each step, the second phase certifier programs are presented with an operation, a supposed answer, and a certification trail for that operation. If the first phase has produced an incorrect output or an incorrect certification trail, then this is detected immediately by the second phase, even before the next operation is processed.

When referring to the time complexity of first and second phase algorithms, we use the notation $(f(n), g(n))$. This is shorthand for saying that the first phase runs in $O(f(n))$ time and second phase runs in $O(g(n))$ time. We use this notation to express both amortized and worse case complexity of certification trail algorithms.

For convenience, we restate the definitions of the operations that we consider in this chapter. The basic operand type is again pairs of the form (it, val) . it is a positive integer, and is called the item number of the pair. Item numbers are used for referring to pairs, and hence all pairs must have distinct item numbers. val can be an arbitrary real number, and is called the item value of the pair. We write $(it, val) < (it', val')$ iff $val < val'$ or $(val = val'$ and $it < it')$. Sets are referred to by a set name, which is assumed to be a positive integer.

1. *makeset*(S): Create the set S and initialize it to be empty. S must not have been used previously in the sequence.

2. $insert((it, val), S)$: Insert the pair (it, val) into the set S . Set S must exist, and item number it must not have been used previously.
3. $delete(it)$: Delete the pair with item number it from whatever set it is currently in. Some pair with item number it must be in some set.
4. $min(S)$: Return the pair with smallest value that is in set S . Set S must exist and may not be empty.
5. $deletemin(S)$: Return the pair with smallest value that is in set S , and then delete it from set S . Set S must exist and may not be empty.
6. $successor((it, val), S)$: Return the smallest pair that is larger than (it, val) that is contained in the set S . If there is no pair larger than (it, val) in S , then return the value nil .
7. $predecessor((it, val), S)$: Return the largest pair that is smaller than (it, val) that is contained in the set S . If there is no pair smaller than (it, val) in S , then return the value nil .
8. $merge(S_1, S_2, S)$: Combine the contents of sets S_1 and S_2 to form set S . Set S_1 and set S_2 must exist, while set S may not have been used previously in the sequence. After this operation sets S_1 and S_2 are destroyed.
9. $split(S, val, S_1, S_2)$: Form the set S_1 by taking all of the pairs in S which are smaller than (∞, val) , and form the set S_2 by taking all of the pairs in S which are greater than (∞, val) . Either S_1 or S_2 may be empty. Set S must exist, and sets S_1 and S_2 must not have been used previously in the sequence. After this operation set S is destroyed.

The operations *successor* and *predecessor* are new, and we sometimes abbreviate them as *succ* and *pred* respectively. Unlike the last chapter, when we perform splittable priority queue operations we maintain the usual semantics of *delete*.

A data structure that supports the operations *makeset*, *insert*, *delete*, *min*, and *deletemin* we call a priority queue. A data structure that supports priority queue operations, and in addition supports the *merge* operation we call a mergeable priority queue. A data structure that supports priority queue operations, and in addition supports the *split* operation we call a splittable priority queue.

Any data structure that supports the operations *makeset*, *insert*, *delete*, *min*, *deletemin*, *predecessor*, and *successor* we call a search data structure. Also, any data structure that supports search data structure operations, and in addition supports

the split operation we call a splittable search data structure. Notice that a splittable search data structure is strictly more powerful than a splittable priority queue.

3.1 Certification of Search Data Structures

Before describing how to certify mergeable priority queues, we discuss certifying search data structures.

In the first phase, each set is represented by its own balanced binary tree. We also maintain the arrays *root* and *where*. The array *root* is indexed by set names, and *root*[*S*] gives a pointer to the root of the balanced binary tree which implements the set *S*. The array *where* is indexed by item numbers, and *where*[*it*] gives a pointer to the node in the balanced binary tree which contains the pair with item number *it*. If *S* is the name of a set which has not been created by a *makeset* operation, then *root*[*S*] will be *nil* by default. If *it* is an item number which is not in use, then *where*[*it*] will be *nil* by default also.

All of the operations *insert*, *delete*, *min*, *deletemin*, *pred*, and *succ* are implemented in the usual way. However, when the operation *insert*((*it*, *val*), *S*) is processed, the value *pred*((*it*, *val*), *S*) is produced as the certification trail. All other operations produce the value *nil* for the certification trail.

The basic data type used to represent the sets in the second phase is an array indexed set of doubly linked lists [76]. For each set created by a *makeset* operation there will be a separate doubly linked list. Each node in a doubly linked list stores a pair contained in the set represented by the linked list. Also, each node contains a field specifying the name of the set which the linked list represents. The nodes in each list will be sorted according to the pairs they contain. We also maintain two arrays. The array *where* is indexed by item numbers, and given an item number *it*, *where*[*it*] points to the node containing the pair with item number *it*. If there is no pair with item number *it* contained in any set, then *where*[*it*] will be *nil*. The array *head* is indexed by set names, and given a set name *S*, *head*[*S*] contains the item number of the smallest pair contained in set *S*. If *S* is empty then *head*[*S*] has the special symbol *empty*, and if *S* has not been created by a *makeset* operation, then *head*[*S*] will be *nil*. See Figure 3.1 for an example.

We now discuss how each operation is implemented by the second phase.

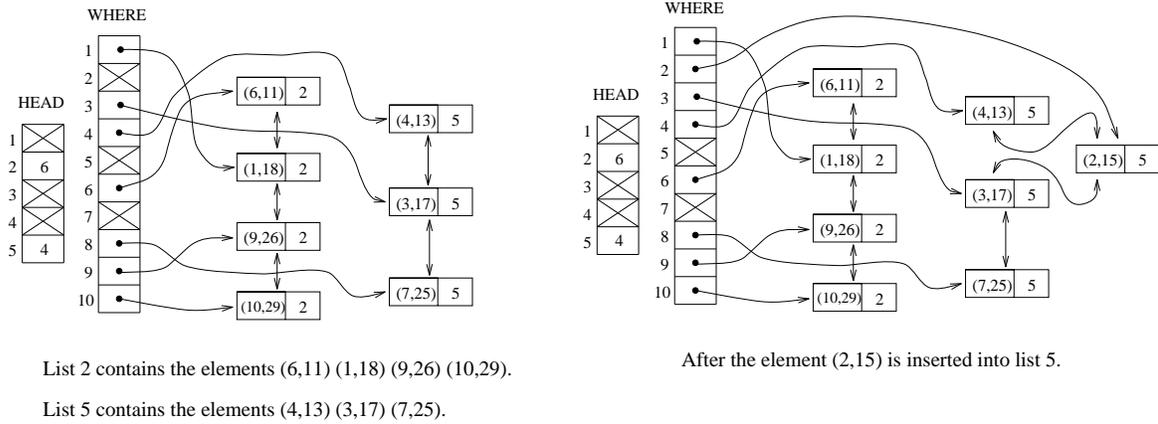


Figure 3.1: Example of an array indexed set of doubly linked lists.

To perform the operation $delete(it)$ the pointer $where[it]$ is traversed and the node pointed to is deleted from the linked list. Both $where[it]$ and $head[S]$ (where S is the set that contains the pair with item number it) are updated as appropriate.

The operation $predecessor((it, val), S)$ is a query operation, and the algorithm needs to check that the supposed answer is correct. If the supposed answer is nil , then the check is simply that (it, val) is smaller than or equal to the pair pointed to by $head[S]$ (in the case that $head[S]$ is $empty$, then quite clearly nil is the correct answer). If the supposed answer is a pair (it', val') , then we perform three checks: (i) $where[it']$ is not nil , (ii) the node pointed to by $where[it']$ stores the pair (it, val) and also is a node for the set S , (iii) $(it', val') < (it, val)$ holds, (iv) the pair in the node after $where[it']$ is greater than (it, val) (assuming that such a pair exists).

The steps performed for the operations min , $deletemin$, and $successor$ are similar to $predecessor$ and are omitted

Now consider the operation $insert((it, val), S)$. Given no additional information, there is no efficient method for the second phase to determine the correct location in the doubly linked list for S in which to insert (it, val) . Thus, the second phase reads the certification trail for this problem. First assume that the certification trail for the operation is a pair (it', val') . Thus (it', val') is supposed to be the predecessor of (it, val) in S . A new node is allocated and the pointer $where[it']$ is used to insert the node into the data structure. However, a few checks need to be performed first. First, we must check that $where[it']$ points to a node which is part

of the list representing set S . Second, we must check that inserting (it, val) after the node pointed to by $where[it']$ leaves the list sorted. If either of these checks fail we output *error*. If both these checks succeed we output *ok*. Then we fill in the fields of node for (it, val) and insert it into the list. Finally, we assign $where[it]$ to be the address of the node for (it, val) .

If our initial assumption is not true and the certification trail contains the value *nil* there are two further cases. If $head[S]$ contains the symbol *empty*, then (it, val) is indeed the smallest element in S , and we simply insert (it, val) into S . Else, $head[S]$ contains the item number of the current smallest element of S , and by using the pointer $where[head[S]]$ we can verify that (it, val) is the smallest element and insert it. In either case, we need to update the value of both $where[it]$ and $head[S]$ as appropriate.

We are left with the following lemma.

Lemma 3.1.1: *Search data structures can be certified in an on-line fashion. Over a sequence of n operations, the first and second phases have a worst-case time complexity of $(\log n, 1)$ per operation.*

The algorithm presented here is a slight generalization of an algorithm presented in [76] which allowed for only one set. The proof of correctness of the generalized version is very similar to the old version, and as no new techniques are involved we omit it. The running time of the first phase and the second phase is apparent from an analysis of the procedures.

3.2 On-Line Checking of Mergeable Priority Queues

In this section we present two schemes for the on-line certification of mergeable priority queues. The first scheme achieves amortized time of $((\log n)A(n), A(n))$ per operation, and worst-case time of $((\log n)^2, \log n)$ per operation. Here, $A(n)$ is the inverse of Ackermann's function, which is an extremely slow growing function [29]. The second scheme achieves amortized time of $(\log n, A(n))$ per operation, and worst-case time of $(\log n, \log n)$ per operation. Thus the first scheme achieves a very good time complexity for the second phase but is unfortunately non optimal in the first phase. The second scheme uses the first scheme as a subroutine and retains the same complexity for the second phase, but makes the first phase optimal.

3.2.1 A First Phase Non-optimal Algorithm

The algorithms in this section are based on the mergeable priority queue structure proposed in [17] by P. van Emde Boas, which itself was based on recommendations of R. E. Tarjan. We will first describe the algorithm as presented in [17], and then discuss how the algorithms for the first phase and the second phase are constructed from it.

The algorithm represents mergeable priority queues using “tritter trees”, which are the basic data structures used in the most efficient implementation of the union-find problem [29]. A mergeable priority queue is represented by a collection of priority queues organized in a tritter tree with a distinct priority queue at each node in the tree. For a mergeable priority queue S , let T_S be the tritter tree for S . Let $root(S)$ be the root node of the tritter tree for S . For every node v in T_S , let $p(v)$ be the parent of v . If v is the root of T_S , then let $p(v) = nil$. For every node v in T_S , let $pq(v)$ be the priority queue stored at that node. We will also maintain the following property for every mergeable priority queue S throughout the course of the algorithm: for every non root node v in T_S , the smallest pair in $pq(v)$ will also be contained in $pq(p(v))$. See Figure 3.2 for an example. Consider a pair (it, val) in some mergeable priority queue S . Because of the property just mentioned and how we perform the operations on a mergeable priority queue, it is the case that for every pair (it, val) there is a sequence of nodes v_1, v_2, \dots, v_m and a j such that: (i) v_i is the parent of v_{i-1} for all $1 < i \leq m$, (ii) v_m is the root of T_S , and (iii) the pair (it, val) is stored in the priority queues at exactly the nodes v_1, v_2, \dots, v_j . Furthermore, for every node v_i with $1 \leq i < j$, (it, val) must be the smallest pair in $pq(v_i)$. If $j \neq m$ then (it, val) is not the smallest pair in $pq(v_j)$. If $j = m$ then (it, val) may or may not be the smallest pair in $pq(v_j)$. Finally, we maintain an array *where* which is indexed by item numbers, so that given an item number it , *where*[it] is a pointer to the node in $pq(v_i)$ which stores the pair with item number it . If there is no pair with item number it in any set, then *where*[it] is *nil* by default.

Procedure *Tritter-Tree-MPQ*:

For each operation, execute the appropriate case:

min(S):

Output the minimum value in $pq(root(S))$.

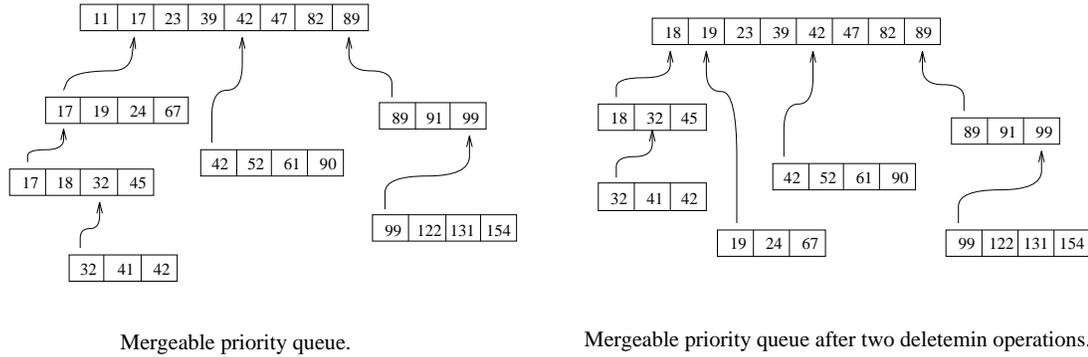


Figure 3.2: Example of tritter tree representation of MPQ.

insert((*it*, *val*), *S*):

Insert the pair (*it*, *val*) into $pq(\text{root}(S))$, and then update the array *where*.

merge(S_1, S_2, S):

Assume that the number of pairs contained in S_1 is less than or equal to the number of pairs in S_2 .

Insert the smallest pair in S_1 into $pq(\text{root}(S_2))$, and make T_{S_1} a child of the root of T_{S_2} .

delete(*it*):

Let v_0 be the node containing the priority queue pointed into by *where*[*it*].

Let *val* be the item value for the pair with item number *it*.

Set *where*[*it*] = *nil*.

Set $i = 0$.

WHILE ((*it*, *val*) is the smallest pair in $pq(v_i)$) AND ($p(v_i) \neq \text{nil}$) DO

Delete (*it*, *val*) from the priority queue $pq(v_i)$.

Set $i = i + 1$.

Set $v_i = p(v_{i-1})$.

END WHILE

Delete (*it*, *val*) from the priority queue $pq(v_i)$.

FOR $j = 0$ TO $i - 1$ DO

Insert the smallest pair in $pq(v_j)$ into $pq(v_i)$, and make v_j a child of v_i .

(* Note, this operation is analogous to performing a path compression operation in the normal union-find data structure. *)

END FOR

We do not describe how the operation *deletemin*(*S*) is performed, but instead

note that it can be simulated by performing a *min* followed by a *delete* operation.

In [17], the authors show that a sequence of n mergeable priority queue operations takes $O(nA(n)T(n))$ time, where $T(n)$ is the time needed to perform a priority queue operation. Also, the worst-case time to process any individual operation is $O(T(n) * \log(n))$. This is because the longest path which can be formed in a tritter tree after n operations is $O(\log n)$. This leads us to the following lemma.

Lemma 3.2.1: *Mergeable priority queues can be certified in an on-line fashion. Over a sequence of n operations, the first and second phases have an amortized time complexity of $((\log n)A(n), A(n))$ per operation, and a worst-case time complexity of $((\log n)^2, \log n)$ per operation.*

Proof: Both phases represent the mergeable priority queues using the general tritter tree scheme just presented. By representing the priority queues at the nodes of the trees as search data structures and using the certification scheme presented in the last section, we achieve the stated time bounds. ■

3.2.2 A First Phase Optimal Algorithm

To achieve an algorithm for the first phase which has optimal time complexity, we devise a new type of mergeable priority queue data structure, which we call an *H-heap*.

The tritter tree type mergeable priority queue presented in the last section was originally developed to be used in conjunction with a range-restricted priority queue developed by P. van Emde Boas [17]. Specifically, when the elements stored are integers in the range $[1, 2, \dots, M]$, his range-restricted priority queue runs in worst-case $O(\log \log M)$ time per operation. (Actually, his “priority queue” supports the more general search data structure operations.) Using his range-restricted priority queue in conjunction with the tritter tree mergeable priority queue presented in the last section, yields a range-restricted mergeable priority queue operating on the integers in the range $[1, 2, \dots, M]$ that runs in amortized time of $O((\log \log M)A(M))$ per operation, and worst-case time of $O((\log \log M) \log M)$ time per operation. $A(M)$ is again the inverse of Ackermann’s function. Using his range-restricted priority queue in conjunction with the *H-heap* mergeable priority queue structure presented in this section, yields a range-restricted mergeable priority queue operating on the integers

in the range $[1, 2, \dots, M]$ that runs in amortized time of $O(\log \log M)$ per operation, and worst-case time of $O(\log M)$ time per operation. See [21] for complete details.

We now present some notation and definitions which will help us to describe the data structure used to represent a priority queue. Let U be the universe of elements. We use $T(U)$ to denote the set of all sets whose elements are from U . We use $P(T(U))$ to denote the set of all sets whose elements are from $T(U)$. Finally, we use $M(P(T(U)))$ to denote the set of all sets whose elements are from $P(T(U))$.

If T is in $T(U)$ then we write $x = \min(T)$ if and only if $x \in T$, and $\forall y \in T$ we have $x \leq y$. If P is in $P(T(U))$ then we write $x = \min(P)$ if and only if $x \in P$, and $\forall y \in P$ we have $\min(x) \leq \min(y)$. If M is in $M(P(T(U)))$ then we write $x = \min(M)$ if and only if $x \in M$, and $\forall y \in M$ we have $\min(x) \leq \min(y)$. Note that \min on an element of $P(T(U))$ returns an element of $T(U)$, while \min on an element of $M(P(T(U)))$ returns an element of $P(T(U))$. If P is in $P(T(U))$ then we define $\min^*(P)$ to be $\min(\min(P))$. If M is in $M(P(T(U)))$ then we define $\min^*(M)$ to be $\min(\min(\min(M)))$.

We now define an *H-heap*, which represents one priority queue. An H-heap S consists of three structures: $\text{tiny}(S)$ which contains an element of $T(U)$, $\text{pq}(S)$ which contains an element of $P(T(U))$, and $\text{meld}(S)$ which contains an element of $M(P(T(U)))$. These three structures partition the elements of S . In addition, let $C(S)$ and $B(S)$ be two integer fields associated with S . $C(S)$ is the number of times an element of $T(U)$ has been moved from $\text{tiny}(S)$ to $\text{pq}(S)$. We call such an event a *full circle* event. $B(S)$ is used only in the analysis, and contains the number of times “expensive” *delete* operations have been performed on S . See Figure 3.3 for an illustration of an H-heap. In this figure, points represent elements of U , circles represent elements of $T(U)$, triangles represent elements of $P(T(U))$, and the rectangle represents an element of $M(P(T(U)))$. In the following algorithm, n is assumed to be a variable which is an integer containing how many operations have been performed in the sequence thus far.

Procedure *H-heap*:

For each operation execute the appropriate case:

makeset(S):
 Initialize $\text{tiny}(S)$, $\text{pq}(S)$, and $\text{meld}(S)$ to be $\{\}$.

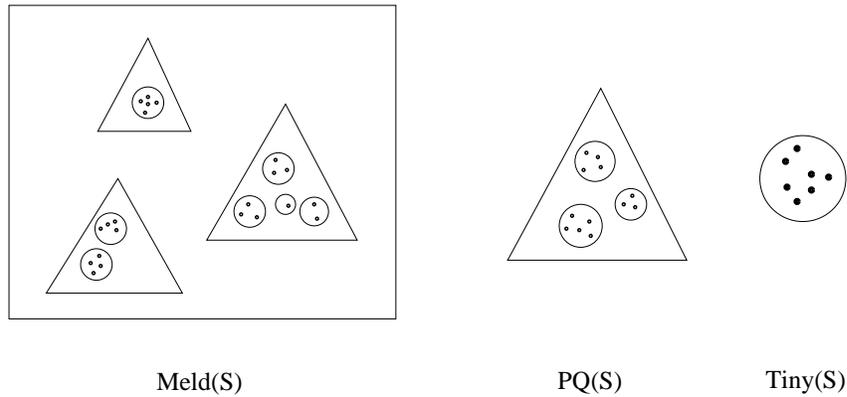


Figure 3.3: Illustration of an H-heap.

Set $B(S) = 0$ and $C(S) = 0$.

insert((*it*, *val*), *S*):

Set $tiny(S) = tiny(S) \cup \{(it, val)\}$.

IF $|tiny(S)| \geq g(n)$ THEN (* $g(n)$ is defined on the following page *)

Set $pq(S) = pq(S) \cup \{tiny(S)\}$.

Set $tiny(S) = \{\}$.

Set $C(S) = C(S) + 1$.

merge(S_1, S_2, S): (* See Figure 3.4 *)

WLOG, assume $C(S_1) \leq C(S_2)$.

Set $tiny(S) = tiny(S_1) \cup tiny(S_2)$.

Set $C(S) = C(S_1) + C(S_2)$, and $B(S) = B(S_1) + B(S_2)$.

Set $pq(S) = pq(S_2)$.

Set $meld(S) = meld(S_2) \cup (meld(S_1) \cup \{pq(S_1)\})$.

IF $|tiny(S)| \geq g(n)$ THEN

Set $pq(S) = pq(S) \cup \{tiny(S)\}$.

Set $tiny(S) = \{\}$.

Set $C(S) = C(S) + 1$.

delete(*it*):

Let *val* be the item value associated with *it*.

Assume (*it*, *val*) is stored in mergeable priority queue *S*.

IF (*it*, *val*) $\in tiny(S)$ THEN

Set $tiny(S) = tiny(S) - \{(it, val)\}$.

ELSE IF (*it*, *val*) $\in pq(S)$ THEN

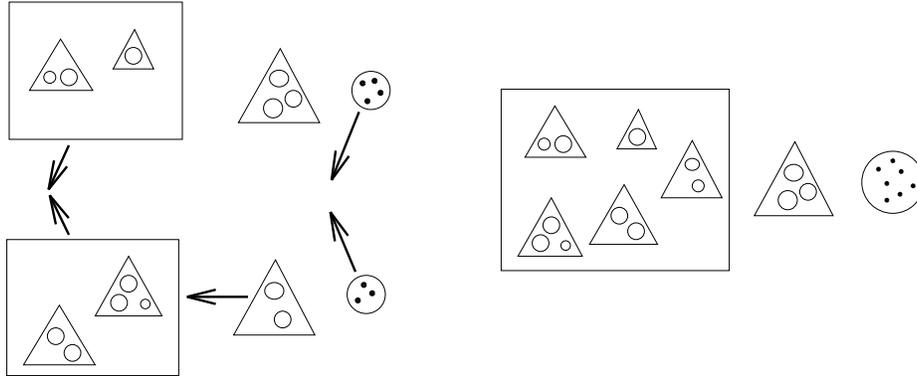


Figure 3.4: Merging two H-heaps.

Let T be the element of $pq(S)$ which contains (it, val) .
 Set $pq(S) = (pq(S) - \{T\}) \cup \{T - \{(it, val)\}\}$.
 ELSE (* $(it, val) \in meld(S)$ *)
 Set $B(S) = B(S) + 1$.
 Let P be the element of $meld(S)$ which contains (it, val) .
 Let T be the element of P which contains (it, val) .
 Let $T' = T - \{(it, val)\}$, and let $P' = P - \{T\}$.
 Set $pq(S) = pq(S) \cup \{T'\}$.
 Set $meld(S) = (meld(S) - \{P\}) \cup \{P'\}$.

A description of *deletemin* is omitted. We note that *deletemin* operation can be simulated by performing a *min* followed by a *delete*.

A careful analysis of algorithm H-heap shows it to be correct. Also, we note that at most a constant number of operations are needed on the $T(U)$, $P(T(U))$, and $M(P(T(U)))$ sets to process any MPQ instruction.

Lemma 3.2.2: *Let $g(n) = \log^2 n$, and assume we have performed n operations. Then the following three assertions hold:*

1. Steps 3-5 of *insert* and 7-9 of *merge* are performed at most $O(n/\log^2 n)$ times.
2. Steps 9-14 of *delete* are performed at most $O(n/\log n)$ times.
3. Step 5 of *merge* will involve a non empty *meld* structure at most $O(n/\log n)$ times.

Proof: We now prove the first assertion. Recall, every time either steps 3-5 of *insert* or steps 7-9 of *merge* are executed, we say that a full circle event has

occurred. Let $N = n/\log^2 n$, and consider the first N operations. Clearly, there can be at most N full circle events over the first N operations. Next, consider the full circle events that occur after the N th operation. For each such full circle event, there are at least $\log^2 N = \log^2 n - o(\log n)$ *insert* operations in the sequence. Thus, there can be at most $O(n/\log^2 n)$ full circle events after the N th operation. So we see that there are at most $O(n/\log^2 n)$ full circle events in all.

We now prove the second assertion. Consider any circle. We claim that this circle can be moved from a *meld* structure to a *Main* structure because of steps 9-14 of *delete* at most $O(\log n)$ times. This is because a circle can be moved from a *Main* structure back to a *meld* structure only by a *Merge* operation, and each time this happened the circle must have been in the priority queue with the smaller C value. So we see that every such *merge* causes the C value of the priority queue containing the circle to at least double, and since the maximum C value of a priority queue is n , our claim is proved. By the first assertion, there are at most $O(n/(\log n)^2)$ full circle events (and hence circles), and since for each full circle event steps 9-14 of *delete* can be executed at most $O(\log n)$ times, our second assertion is proved.

We omit a proof of the third assertion. We claim that any function which bounds the number of times steps 9-14 of *delete* are executed, also bounds the number of times step 5 of *merge* will involve a non empty *meld* structure. ■

Both the first phase and the second phase are based very strongly on algorithm H-heap, with both of them using the same *meld*, *pq*, and *tiny* partitioning of the elements. As in the above lemma, we choose $g(n) = (\log n)^2$. In both phases, the $M(P(T(U)))$ sets are represented as Binomial Queues (or any other optimal MPQ algorithm), and no certification trail is produced for operations on these sets. The first phase represents the $P(T(U))$ sets as search data structures, and we use the methods of the first section of this chapter to certify the operations on them. Finally, we use the tritter tree MPQ certification scheme to represent the $T(U)$ sets.

Thus, we can see that the first phase runs in a total of $O((\frac{n}{\log n} \log n) + (n \log n) + (n(\log \log n)A(n)))$ time. The $(\log \log n)A(n)$ factor of the last term comes from the fact that the $T(U)$ sets contain at most $O(\log^2 n)$ elements, and hence the operations performed on them take only $O((\log \log n)A(n))$ time. Also, it is clear that the first phase has a worst-case time complexity per operation of $O(\log n)$.

Now consider the second phase. Clearly, the second phase spends at most

$O(\frac{n}{\log n}) \log n = O(n)$ time on the $M(P(T(U)))$ sets. Also, the second phase spends at most $O(n)$ time on the $P(T(U))$ sets as we use the search data structure certification scheme of the previous section. Finally, by the results on the tritter tree MPQ certification scheme, we see that the second phase spends $O(nA(n))$ time performing operations on the $T(U)$ structures. As in the first phase, the worst-case time per operation is $O(\log n)$. We summarize this result in the following lemma.

Lemma 3.2.3: *Mergeable priority queues can be certified in an on-line fashion. Over a sequence of n operations, the first and second phases have an amortized time complexity of $(\log n, A(n))$ per operation, and a worst-case time complexity of $(\log n, \log n)$ per operation.*

In order to use *H-heaps* for purpose of constructing range-restricted mergeable priority queues, a slightly more complicated analysis has to be done. For completeness, we present that analysis here. Let the integers in our universe be $[1, 2, \dots, M]$. For the rest of this section, we assume that $g(n)$ is a function with constant value $(\log M)^2$.

The following lemmas are central to our analysis.

Lemma 3.2.4: *Let $f(C(S))$ be an upper-bound on $B(S)$ for all S . Let mergeable priority queue S have been created by the operation $merge(S_1, S_2, S)$, and assume $C(S_1) \leq C(S_2)$. Then $B(S) \leq f(C(S_1)) + f(C(S_2)) + \min(C(S_1), M)$.*

Proof: Each time a full circle event occurs (that is, steps 3-5 of *insert* or steps 7-9 of *merge* are executed), we can associate an unique identifier to the set of type $T(U)$ in the *tiny* structure that is moved to the *pq* structure. Thus, even though the contents of a set may change over time, we still view it as the same set for the purpose of analysis. Now note that $B(S) = B(S_1) + B(S_2) + B$ where B is the number of times that steps 9-14 of *delete* are executed on S . Each time either steps 3-5 of *insert* or 7-9 of *merge* are executed, a set of type $T(U)$ must be moved from *meld*(S) to *pq*(S). Let b_1 be the number of sets moved that were originally from S_1 and let b_2 be the number of sets moved that were originally from S_2 . It is clear that $B = b_1 + b_2$ since even if enough *insert* operations are performed on S for step 2 of *insert* to be executed, the set in *tiny*(S) is moved to *pq*(S) and not *meld*(S). Also, $b_1 \leq C(S_1)$ and $b_1 \leq M$ must hold. However, we claim that steps 9-14 of *delete*

could have been executed b_2 additional times on S_2 . This is because the sets that were originally from S_2 that are moved from $meld(S)$ to $pq(S)$ must have been in $meld(S_2)$ prior to the operation $merge(S_1, S_2, S)$ since we assumed $C(S_1) \leq C(S_2)$. Let S'_2 be the mergeable priority queue resulting from performing the b_2 additional *deletemin* operations on S_2 . We see that $B(S'_2) = B(S_2) + b_2$, and that $C(S'_2) = C(S_2)$. Thus,

$$B(S) = B(S_1) + b_1 + B(S_2) + b_2 = B(S_1) + B(S'_2) + b_1 \leq$$

$$f(C(S_1)) + f(C(S'_2)) + \min(C(S_1), M) = f(C(S_1)) + f(C(S_2)) + \min(C(S_1), M) \blacksquare$$

Lemma 3.2.5: *Assume we have performed n operations. Let $g(n)$ be the constant function $\log^2 M$. Then the following assertions hold:*

1. Steps 3-5 of *insert* and 7-9 of *merge* are executed at most $O(n/\log^2 M)$ times.
2. Steps 9-14 of *delete* are executed at most $O((2n \log M + n - M)/(\log^2 M))$ times.
3. Step 5 of *merge* operates on a non empty *meld* structure at most $O((2n \log M + n - M)/(\log^2 M))$ times.

Proof: The first assertion is obvious.

Let S'_1, \dots, S'_y be all of the mergeable priority queues active after the n th operation is performed. By the term “active” we mean all of the queues that have been created by either a *makequeue* or *merge* operation but have not been destroyed by being the first or second operand to a *merge* operation. Then the sum $\sum_{i=1}^y B(S'_i)$ is exactly the number of times steps 9-14 of *delete* have been executed. Now notice that the quantity $\log^2 M \sum_{i=1}^y C(S'_i)$ is a lower bound on the number of *insert* operations executed in the sequence.

We first prove the following claim: for any priority queue S active during the sequence of operations, if $C(S) \leq M$ then $B(S) \leq C(S) \log C(S)$. There are two cases. The first case is that S is the product of a *makequeue*(S) operation. It is easy to see then that $B(S) = 0$, and the claim is trivially true. The second case is that S is the product of an operation of the form $merge(S_1, S_2, S)$. Without loss of generality we assume that $C(S_1) \leq C(S_2)$, else we reverse the roles of S_1 and S_2 in the following analysis. By induction we assume that $B(S_1) \leq C(S_1) \log C(S_2)$, and that $B(S_2) \leq C(S_2) \log C(S_2)$. Thus from lemma 3.2.4 we see that

$$B(S) \leq C(S_2) \log C(S_2) + C(S_1) \log C(S_1) + C(S_1).$$

Let $X = C(S_1)$, and let $a = C(S_2)/C(S_1)$. Thus $C(S_2) = aX$ and $a \geq 1$. Rewriting the above we get

$$\begin{aligned}
B(S) &\leq aX \log(aX) + X \log X + X = \\
&aX \log \frac{aXa(a+1)}{a(a+1)} + X \log \frac{Xa(a+1)}{a(a+1)} + X = \\
&aX(\log((a+1)X) + \log \frac{a}{a+1}) + X(\log((a+1)X) + \log \frac{1}{a+1}) + X = \\
&(a+1)X \log((a+1)X) + aX \log \frac{a}{a+1} + X \log \frac{1}{a+1} + X = \\
&(a+1)X \log((a+1)X) + X(1 - (a \log \frac{a+1}{a} + \log(a+1))).
\end{aligned}$$

Since $a \geq 1$, $a \log \frac{a+1}{a} \geq 0$, and $\log(a+1) \geq 1$ we get $X(1 - (a \log \frac{a+1}{a} + \log(a+1))) \leq 0$. Thus, we get $B(S) \leq C(S) \log C(S)$ and the claim is proved.

Next, using induction we show that for any priority queue S active during the sequence of operations, $B(S) \leq 2C(S) \log M + C(S) - M$. As above, the critical case is when S is the product of a *merge*(S_1, S_2, S) operation. Again, we can assume that $C(S_1) \leq C(S_2)$. There are four cases. The first case is that $C(S_1) + C(S_2) \leq M$. This case is handled by the claim above. The second case is that $M \geq C(S_2) \geq C(S_1)$, but that $C(S_1) + C(S_2) \geq M$. Again using lemma 3.2.4 we see that

$$\begin{aligned}
B(S) &\leq C(S_2) \log C(S_2) + C(S_1) \log C(S_1) + C(S_1) \leq \\
&(C(S_1) + C(S_2)) \log M + C(S_1) \leq \\
&2(C(S_1) + C(S_2)) \log M \leq \\
&2(C(S_1) + C(S_2)) \log M + C(S_1) + C(S_2) - M.
\end{aligned}$$

The third case is that $C(S_2) \geq M \geq C(S_1)$. Again using lemma 3.2.4, we see that

$$\begin{aligned}
B(S) &\leq 2C(S_2) \log M + C(S_2) - M + C(S_1) \log C(S_1) + C(S_1) \leq \\
&2(C(S_2) + C(S_1)) \log M + C(S_1) + C(S_2) - M.
\end{aligned}$$

The fourth case is $C(S_2) \geq C(S_1) \geq M$. With one final application of lemma 3.2.4 we see that

$$B(S) \leq (2C(S_1) \log M + C(S_1) - M) + (2C(S_2) \log M + C(S_2) - M) + M =$$

$$2(C(S_1) + C(S_2)) \log M + C(S_1) + C(S_2) - M.$$

This completes the proof of the second claim. Since the number of insertions is a lower bound on the number operations in the sequence, the second assertion is proved.

We omit a proof of the third assertion. We claim that any function which bounds the number of times steps 9-14 of *delete* are executed also bounds the number of times that step 5 of *merge* involves a non empty *meld* structure. ■

With this analysis, we see that by implementing $M(P(T(U)))$ and $T(U)$ sets as Binomial Queues (or any other optimal mergeable priority queue), and implementing the $P(T(U))$ sets as range-restricted priority queues, we get a new efficient range-restricted mergeable priority queue. See [21] for complete details.

3.3 On-line Checking of Splittable Search Data Structures

We now consider the problem of checking a sequence of splittable search data structure (SSDS) operations in an on-line fashion.

We now describe the algorithms $CertSSDS_1$ and $CertSSDS_2$.

Procedure $CertSSDS_1$: we represent the elements of a set S as a two-three tree. The leaves of the two-three tree in left to right order correspond to the elements of S in sorted order. The operations *insert*, *delete*, *deletemin*, *min*, *predecessor*, *successor*, and *split* are all handled using standard two-three tree operations [1]. We now discuss the certification trail output by $CertSSDS_1$. The operations *min*, *deletemin*, *makeset*, *predecessor*, *successor* and *delete* all have *nil* for a certification trail. The certification trail for the operation $insert((it, val), S)$ is $pred((it, val), S)$. The certification trail for the operation $split(S, val, S_1, S_2)$ is the largest pair in the resulting set S_1 , or *nil* if S_1 is empty.

Procedure $CertSSDS_2$: we represent the contents of all of the sets in a single doubly linked list L . For any set S , the contents of S will occupy a contiguous part of L , and the elements will appear in sorted order. In addition, two markers called $start(S)$ and $end(S)$ appear immediately before and after the elements in L corresponding to the set S . These markers help overcome the major difficulty that $CertSSDS_1$ could fault while processing an operation on the set S , and output a pair as the certification trail that is not even contained in the set S . Finally, we maintain the array *where* indexed by item numbers so that given an item number it , $where[it]$

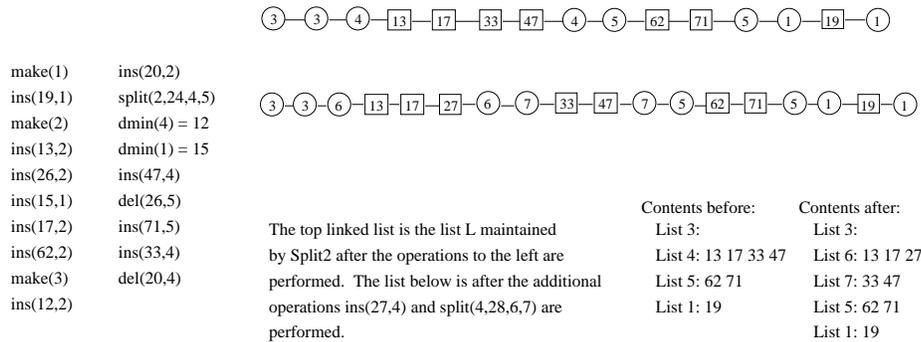


Figure 3.5: Representation of the splittable search data structure for the second phase.

contains a pointer to the node in L which stores the pair with item number it . See figure 5 for an example. For each operation, let ans be the supposed answer and $cert$ the certification trail, and perform the appropriate step.

makeset(S):

Insert the markers $start(S)$ and $end(S)$ at the beginning of L .

insert((it, val), S):

IF ($cert = nil$) THEN

Check that if S is not empty that the smallest element in S is larger than or equal to (it, val).

Insert (it, val) immediately after $start(S)$ in L .

Update the value of $where[it]$.

ELSE

Check that $cert$ is a pair of the form (it', val').

Check that $where[it']$ falls between $start(S)$ and $end(S)$ in L .

Check that val' is correct.

Check that (it, val) is larger than (it', val') and smaller than or equal to the element succeeding (it', val') in S (if that second element exists).

Insert (it, val) in L after (it', val').

Update the value of $where[it]$.

delete(it):

Remove the node pointed to by $where[it]$ from L .

Set $where[it] = nil$.

min(S):

IF $ans = nil$ THEN

Check that $start(S)$ is immediately before $end(S)$.

ELSE

Check that ans is the first element after $start(S)$ in L .

predecessor((it, val), S):

IF $ans = nil$ THEN

Check that the element after $start(S)$ is larger than or equal to (it, val).

ELSE

Check that ans is a pair of the form (it', val').

Check that $where[it']$ falls between $start(S)$ and $end(S)$ in L .

Check that val' is correct.

Check that (it', val') is smaller than (it, val) and that the element

following (it', val') is larger than or equal to (it, val) (if that second element exists).

successor((it, val), S):

IF $ans = nil$ THEN

Check that the element before $end(S)$ is smaller than or equal to (it, val).

ELSE

Check that ans is a pair of the form (it', val').

Check that $where[it']$ falls between $start(S)$ and $end(S)$ in L .

Check that val' is correct.

Check that (it', val') is larger than (it, val) and that the element

preceding (it', val') is smaller than or equal to (it, val).

split(S, val, S_1, S_2):

IF ($cert = nil$) THEN

Check that the element after $start(S)$ is larger than (∞, val).

Insert the markers $start(S_1)$ and $end(S_1)$ immediately before $start(S)$ in L .

Replace the markers $start(S)$ and $end(S)$ with $start(S_2)$ and $end(S_2)$ respectively.

ELSE

Check that $cert$ is a pair of the form (it', val').

Check that $where[it']$ points to an element between
 $start(S)$ and $end(S)$.

Check that val' is correct.

Check that (∞, val) is larger than (it', val') and smaller than the
element succeeding (it', val') in S (if that second element exists).

Replace the marker $start(S)$ by $start(S_1)$ and $end(S)$ by $end(S_2)$.

Insert the markers $end(S_1)$ and $start(S_2)$ immediately after (it', val') in L .

We now discuss the implementation of $CertSSDS_1$ and $CertSSDS_2$. The only difficulty is in $CertSSDS_2$ when we need to perform the check that $where[it']$ falls between $start(S)$ and $end(S)$ in L while performing an *insert* or *split* operation, and that ans is contained in the set S while performing a *predecessor* operation. For

this we use a result of Dietz and Sleator [30] which allows one to perform a sequence of insert and delete operations on a linked list, intermixed with queries of the following form: given pointers to two elements of the linked list, determine the relative order of the elements pointed to. They achieve constant time for both update and query operations on a list. Thus we see that $CertSSDS_1$ requires at most $O(\log m)$ time per operation where m is the number of operations performed so far, and $CertSSDS_2$ requires only $O(1)$ time per operation.

Lemma 3.3.1: *$CertSSDS_1$ and $CertSSDS_2$ certify the on-line splittable priority queue problem. Over a sequence of n operations, $CertSSDS_1$ and $CertSSDS_2$ have a worst-case time complexity of $(\log n, 1)$ per operation.*

Proof: The worst-case running times of both algorithms have already been established. Also, we note that $CertSSDS_1$ always produces the correct answer.

We claim that if $CertSSDS_2$ accepts the first k operations, then after the k th operation is processed, for every “active” set S , the contents of S are stored in sorted order between $begin(S)$ and $end(S)$. Also, for a set S , the markers $start(S)$ and $end(S)$ will be present in the list only if S is currently active. A set S is active if it has been created by either a *makeset* or *split* operation, but has not yet been destroyed by being the first argument to a *split* operation. We prove this statement by induction on k . The statement is clearly true for $k = 1$. Now assume that it is true for $k = i$. So consider the $(i + 1)$ th operation. By assumption, $CertSSDS_2$ does not indicate *error* while processing this operation. In the following discussion, let L be list in $CertSSDS_1$ after the i th operation is processed, and L' the list after the $(i + 1)$ th operation is processed. Also, let *ans* and *cert* be the supposed answer and certification trail respectively for the $(i + 1)$ th operation. There are several cases depending upon the $(i + 1)$ th operation.

- *makeset*(S): By examining the procedure $CertSSDS_2$, we see that the list L' differs from L only by the presence of the markers $start(S)$ and $end(S)$ at the front of L' . Since sets are initialized to be empty, it is clear that the property holds for S in L' . Now consider any set S' that existed before the operation *makeset*(S) was executed. We see that the sublist of L between $start(S')$ and $end(S')$ is the same before and after the operation *makeset*(S) is processed. Since the contents of S' are not changed by the operation *makeset*(S), we see that the property holds for S' in L' .

- *insert* $((it, val), S)$: Let p_1, p_2, \dots, p_m be the sublist of L between $state(S)$ and $end(S)$. There are two cases. The first is that $cert = nil$. We see that the sublist of L' between $start(S)$ and $end(S)$ is $(it, val), p_1, p_2, \dots, p_m$. Also, since we are assuming that $CertSSDS_2$ does not indicate an error, it must be the case that $(it, val) < p_1$ holds, and hence the property holds for S in L' . For any other set S' , we see that the sublist between $start(S')$ and $end(S')$ in L is the same as in L' . Since the contents of a set S' are not changed by the operation $insert((it, val), S)$ if $S' \neq S$, the property must hold for S' in L' as well. The case that $cert \neq nil$ is similar and is omitted.
- *split* (S, val, S_1, S_2) : First note that for any set S' not appearing as an argument to the *split* operation, the sublist for S' in L is the same as in L' , and since the *split* operation does not change the contents of S' , the property is preserved in L' for S' . Also, notice that the set S is destroyed by the *split* operation, and that the markers $start(S)$ and $end(S)$ are not present in L' . There are two cases. First assume that $cert$ is a pair of the form (it', val') . Let p_1, p_2, \dots, p_m be the sublist of L between $start(S)$ and $end(S)$. Thus there is an j such that the sublist of L' between $start(S_1)$ and $end(S_1)$ is p_1, p_2, \dots, p_j and that the sublist of L' between $start(S_2)$ and $end(S_2)$ is $p_{j+1}, p_{j+2}, \dots, p_m$. Since by assumption $CertSSDS_2$ does not indicate an error, we see that $p_j < (\infty, val)$, and that $(\infty, val) < p_{j+1}$ (assuming that $j \neq m$). Thus by the definition of *split*, we see that S_1 should contain the elements p_1, p_2, \dots, p_j and that S_2 should contain the elements $p_{j+1}, p_{j+2}, \dots, p_m$, and we are done with this case. The argument for the case that $cert = nil$ is similar and is omitted.
- *delete* (it) : Let (it, val) be the pair to be deleted. Also, assume that (it, val) is contained in set S . Clearly the property holds for all other sets $S' \neq S$ in the new list L' . Let p_1, p_2, \dots, p_m be the sublist of L between $state(S)$ and $end(S)$. We see that there is a j so that $p_j = (it, val)$, and that the sublist of L' between $start(S)$ and $end(S)$ is $p_1, p_2, \dots, p_{j-1}, p_{j+1}, \dots, p_m$, which is exactly the contents of S after the *delete* operation.
- *pred* $((it, val), S)$, *succ* $((it, val), S)$, *min* (S) : We can immediately see that $L' = L$. Since these three operations are query operation and do not change the contents of any set, we are done.

Now we wish to show that if $CertSSDS_2$ accepts the the first $i - 1$ operations and supposed answers, and also accepts the i th operation, then the i th supposed answer is correct. The only relevant case is when the i th operation is a query operation. The three possibilities for the i th operation are *predecessor*, *successor* and *min*, and as they are very similar we will only discuss the case for *predecessor*. So

let the i th operation be of the form $pred((it, val), S)$. Let p_1, p_2, \dots, p_m be the sublist of L between $start(S)$ and $end(S)$. The first case is that ans is a pair of the form (it', val') . It is clear that there must be a j so that $p_j = (it', val')$. Also, we see that $p_j < (it, val) < p_{j+1}$. Since by the claim above, we have that p_1, p_2, \dots, p_m are precisely the contents of S after the i th operation, and that $p_1 < p_2 < \dots < p_m$ also holds, by the definition of *predecessor* we see that ans is correct. The second case is that $ans = nil$. The proof is very similar to the case that $ans \neq nil$, and we omit it.

Finally, we need to show that if $CertSSDS_2$ indicated *ok* over the first $i - 1$ operations, and then for the i th operation is given the input, the correct output, and also the correct certification trail produced by $CertSSDS_1$, then $CertSSDS_2$ accepts the i th operation as well. Again, we need to consider every possible operation. The proof techniques needed are similar to those used above, so we do not present the details. ■

Chapter 4

On-Line Certification of Approximate Nearest Neighbor Queries

In this chapter we show how to efficiently certify approximate nearest neighbor queries. We define the approximate nearest neighbor problem as follows. Given a set of points in d -dimensional space, we wish to preprocess the points so that given a query point q and an $\epsilon > 0$, we can quickly find a point p in the set, such that $\text{dist}(q, p) \leq \text{dist}(q, p_{opt}) * (1 + \epsilon)$, where p_{opt} is the actual nearest neighbor of q in the set. See Figure 4.1 for an example in two dimensions. In this figure, q is the query point and $\epsilon = .5$. The point a is the actual nearest neighbor of q . The outer circle has a radius of 1.5 times that of the inner circle, so we see that either of the points labeled b or c would also be acceptable answers to the approximate nearest neighbor query.

We now describe a general method for preprocessing a multi-dimensional point set. We assume that no two points have the same coordinate for any axis. The basic idea is to construct a multi-dimensional search tree [65]. Each node v in the tree has the labels $Outer(v)$ and $Split(v)$, which we define shortly. Each leaf node of the tree stores exactly one point in the data set. Let u be a node, which will be the root of the tree. We associate all of the points in the data set with u . Let $Outer(u)$ be any d -dimensional box which contains all of the points.

We recursively perform the following procedure until the number of points associated with a subtree is one. Choose a hyperplane H which splits $Outer(u)$ into two smaller rectangles, R_1 and R_2 . H is chosen so that none of the points in $Outer(u)$ lie in H . Set $Split(u) = H$. Let v and w be two new nodes, and make v and w children of u . Set $Outer(v) = R_1$, and associate with v all of the points associated with u that lie inside R_1 . Set $Outer(w) = R_2$, and associate with w all of the points associated with u that lie inside R_2 . Then recurse on v and w .

At each leaf node of the tree, we store the coordinates of the point which is associated with that leaf when the recursion terminates. In addition, each internal node v will have a label $Inner(v)$, which is the smallest d -dimensional rectangle that contains all of the points stored in the subtree rooted at v . These labels are most easily calculated in a bottom-up fashion. Clearly, for all internal nodes in the tree we have that $Outer(v)$ contains $Inner(v)$ (but not necessarily properly).

There are many different methods for choosing the splitting hyperplane at each internal node of a multi-dimensional search tree. The splitting strategy chosen affects the structure of the tree, and also the running time of query procedures on the

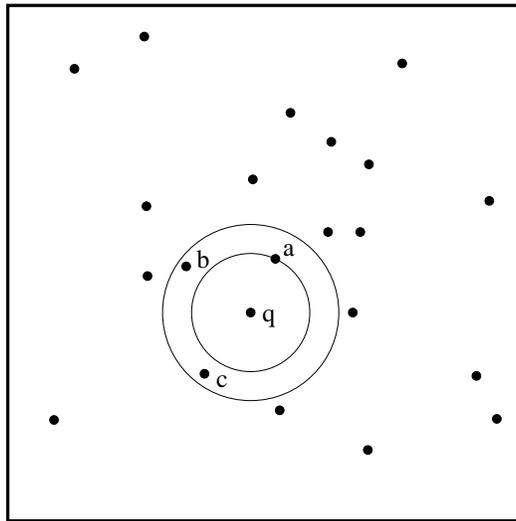


Figure 4.1: Example of approximate nearest neighbor query.

tree. When a particular splitting strategy produces a highly unbalanced tree, it may be necessary to artificially balance the tree using a centroid decomposition [44] or some other technique. Such restructuring is necessary in only unusual circumstances. However, while different splitting strategies can lead to different running times for the query algorithms, the correctness of a query algorithm does not, in general, depend on the splitting strategy chosen. Some methods for querying the tree rely heavily upon the *Inner* labels on the internal nodes. Others do not, and for these we do not need to compute the *Inner* labels.

A theoretically interesting method for choosing the splitting hyperplane at each internal node was developed by Callahan and Kosaraju [23], and the resulting decomposition is called the *fair split tree*. A fair split tree can be built in $O(n \log n)$ time on a data set containing n points. The fair split tree was originally developed to help solve the all k -nearest-neighbors problem, and also the n -body potential field problem. Callahan has also shown how this data structure is useful for performing approximate nearest neighbors queries efficiently.

4.1 Calculating Approximate Nearest Neighbors

We are given a set of points in d -dimensional space. We wish to preprocess the points to handle (q, ϵ, k) -queries on the set of points, which are defined as follows. q is a d -dimensional query point, ϵ is a real number such that $\epsilon > 0$, and k is an integer such that $k \geq 1$. We want to find a subset $P = \{p_1, p_2, \dots, p_k\}$ of the set of points such that if i is the index of the point in P farthest from q , then $\text{dist}(q, p_i) \leq \text{dist}(q, p_{opt}^k) * (1 + \epsilon)$, where p_{opt}^k is the actual k th nearest point to q .

The following notation is useful. For any set of nodes X in a multi-dimensional search tree, let

$$P_X = \{p \mid p \text{ is a point stored in a leaf that is a descendant of a node in } X \}.$$

Given a query, the method of Callahan outputs a set of nodes A in the tree with the following four properties:

1. A is constant sized. More precisely, the value of $|A|$ is bounded by a function dependent on both ϵ and d , but not k or n .
2. No node in A is an ancestor or a descendant of another node in A .
3. $|P_A| \geq k$ holds.
4. Any subset of P_A of size k is a correct answer to the (q, ϵ, k) -query.

Callahan's query algorithm runs in $O(\log n)$ time, where n is the number of points stored in the fair split tree. The constants in the query algorithm have an exponential dependence on ϵ and the dimension, though not on the value of k . The independence of the running time with regard to the value of k is possible because only the constant sized set of nodes A is output.

4.2 Checking The Correctness

In the second phase, we are given an (q, ϵ, k) -query, and a supposed answer A to the query. Recall that A is represented as a set of nodes in the fair split tree. We assume that the second phase has a local copy of the fair split tree. We wish to check that the set A has the four properties outlined in the last section.

Let S_o be the smallest sphere centered at q which encloses all of the *Inner* rectangles for the nodes in A . Let S_i be the smallest sphere centered at q , such that if A' is the subset of those nodes in A whose *Inner* rectangles intersects S_i , then $|P_{A'}| \geq k$ holds.

We check that the following conditions are satisfied:

1. A is constant sized.
2. No node in A is an ancestor or a descendant of another node in A .
3. $|P_A| \geq k$ holds.
4. The radius of S_o is less than or equal to $1 + \epsilon$ times the radius of S_i .
5. The only points that lie inside S_i are contained in P_A .

The fifth condition, together with the definition of the sphere S_i , implies that the k th closest point to q can not properly lie inside S_i . In other words, the radius of S_i is a lower bound on the distance from the k th closest point to q . The third condition implies that all of the points in P_A are distinct, and taken with the fourth condition we see that any k points in A are an acceptable answer to the approximate nearest neighbor query.

Since the number of nodes in A is of constant size, the second phase can in constant time construct the spheres S_i and S_o . To check the third condition, we assume that lowest common ancestor preprocessing has been performed on the tree (such preprocessing can be performed in $O(n)$ time [71]). Then $|A| * |A|$ lowest common ancestor queries can verify that no node in A is an ancestor of another node in A (and hence no node in A is a descendant of another node in A). The second condition can be checked in constant time if we augment the fair split tree to contain the size of the subtree at each internal node. The fourth condition is trivial to check in constant time. The fifth condition is the only one which is difficult to check. To perform this check, we will use a certification trail. Let *Cube* be the smallest isothetic d -dimensional cube which contains S_i . The certification trail will consist of a constant sized set of nodes C in the fair split tree which have the following properties:

1. None of the *Inner* rectangles for the nodes in C has a non-trivial intersection with S_i .

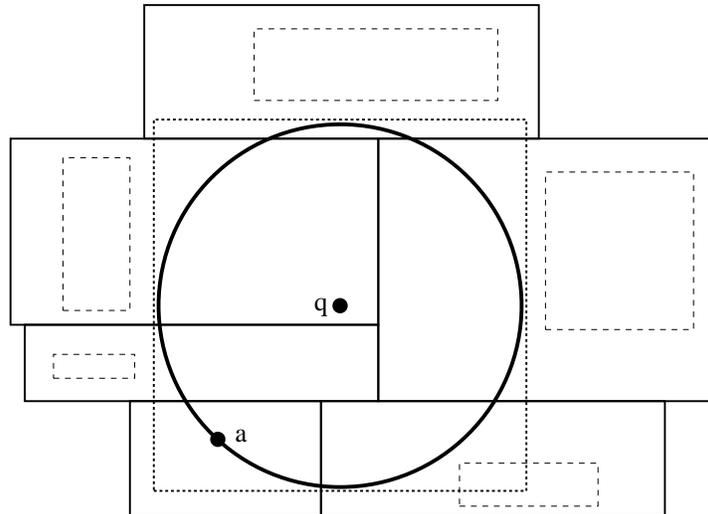


Figure 4.2: Example of covering the box around the sphere.

2. The *Outer* rectangles for the nodes in C and A cover *Cube*.
3. None of the *Outer* rectangles for the nodes in A or C has a non-trivial intersection with any of the other *Outer* rectangles for the nodes in A or C .

These checks suffice because if a point lies inside the *Outer* rectangle of a node, then by definition it must also lie inside the *Inner* rectangle for that node. See Figure 4.2 for an example. In this figure, q is the query point, we assume $k = 1$, and the point a is the supposed answer. The solid box around a is the *Outer* rectangle for the leaf which stores a . The large dotted box around the circle is completely covered by the *Outer* rectangles, which are drawn with solid lines. The *Inner* rectangles are drawn with dashed lines, and we see that none of them has a non-trivial intersection with the circle. Since the point a is stored in a leaf, the *Inner* rectangle associated with the *Outer* rectangle is degenerate and contains only the point a .

Given that C is of constant size, the first condition is easy to check in constant time. Since A is also of constant size, the third condition can be checked using $(|A| + |C|)^2$ lowest common ancestor queries. For the second condition, we note that it would be sufficient to check that the *Outer* rectangles cover S_i , but that is a more difficult condition to check than the one we present. We are able to use a simple volume metric approach to determine if the *Outer* rectangles for the nodes in A and C completely cover *Cube*. We sum the volumes of the intersections between *Cube* and

each *Outer* rectangle for the nodes in A and C . Because all of the *Outer* rectangles for the nodes in A and C are disjoint, the second condition is satisfied if and only if this sum equals the volume of *Cube*. To avoid roundoff and overflow errors, it is possible to sort and rescale all of the coordinates of the *Outer* rectangles and also *Cube* so that the coordinates are all integers. Since there are only a constant number of rectangles and hence only a constant number of coordinates, it will be the case that the rescaled coordinates are all small valued. Thus we see that the arithmetic for the volume computations can be performed inside a machine word, and hence all of the volume computations will only take a constant amount of time. The correctness of the rescaling procedure is ensured by the following technical lemma.

Lemma 4.2.1: *Let r_1, r_2, \dots, r_n be a set of d -dimensional rectangles, with each rectangle r_i represented as a Cartesian product $[r_i(x_1), r_i(x'_1)] \times [r_i(x_2), r_i(x'_2)] \times \dots \times [r_i(x_d), r_i(x'_d)]$ in \mathfrak{R}^d . Let $1 \leq j \leq d$ be an arbitrary dimension. The numbers $r_i(x_j)$ and $r_i(x'_j)$ are referred to as the coordinates of r_i in the j th dimension. Let B be the set of the coordinates of all of the rectangles in the j th dimension. (So, for each r_i , both $r_i(x_j)$ and $r_i(x'_j)$ appear in B .) Let $b_1 < b_2 < \dots < b_m$ be the elements in B in sorted order, with duplicates removed. Let k be an index with $1 \leq k \leq m$, and let l be a real number such that $b_{k-1} < l < b_{k+1}$. (As a notational convenience, we define $b_0 = -\infty$ and $b_{m+1} = \infty$.) Construct the rectangles r'_1, r'_2, \dots, r'_n from r_1, r_2, \dots, r_n as follows: r'_i has the same coordinates as r_i except in the j th dimension, where a coordinate with value b_k is replaced with value l .*

Then, the rectangles r_2, r_3, \dots, r_n cover the rectangle r_1 if and only if the rectangles r'_2, r'_3, \dots, r'_n cover r'_1 .

Proof: Let $j, B, b_1 < b_2 < \dots < b_m, k$ and l be as in the lemma. We define a function f as follows. Let $p = (p_1, p_2, \dots, p_n)$ be a point \mathfrak{R}^d . If $p_j \leq b_{k-1}$ or $b_{k+1} \leq p_j$, then $f(p) = p$. Else, if $b_{k-1} < p_j < b_{k+1}$, then there are three further cases. First, if $p_j = b_k$, then $f(p) = (p_1, p_2, \dots, p_{j-1}, l, p_{j+1}, \dots, p_d)$. If $p_j < b_k$, then $f(p) = (p_1, p_2, \dots, p_{j-1}, (b_{k-1} + l)/2, p_{j+1}, \dots, p_d)$. If $p_j > b_k$, then $f(p) = (p_1, p_2, \dots, p_{j-1}, (l + b_{k+1})/2, p_{j+1}, \dots, p_d)$.

It can be shown that the function f is a one-to-one and onto mapping from the points in r_1 to the points in r'_1 . Consider a point p in r_1 , and its image p' in r'_1 . It can be shown for $j > 1$ that p is contained in r_j if and only if p is contained in r'_j . Since f is onto, this suffices to prove the theorem. ■

The existence of a constant sized certification trail C can be demonstrated by a careful analysis of the query method of Callahan. We will not present the details here, but will instead note that the certification trail C can be found by the query algorithm without any asymptotic loss in efficiency.

In order for this checking algorithm to be performed, the second phase needs a copy of the fair split tree. Either it can construct its own copy in $O(n \log n)$ time, or it can receive as input a copy of the tree that was constructed by the first phase and then check in $O(n)$ time that it is a proper multi-dimensional search tree. This check can be performed as follows. First, the second phase needs to check that each point in the input set is contained in exactly one leaf of the tree. This can be done efficiently if the second phase receives a certification trail giving the correspondence between the leaves and the input points. Then the second phase can use a bottom-up procedure to check that the *Inner* labels are correct, and a top-down procedure to check that the *Outer* labels are correct. A careful analysis of the second phase algorithm shows that it is not necessary for the second phase to specifically check that the tree it is given is the fair split tree. If the tree it is given is a proper multi-dimensional search tree, that is sufficient to guarantee that if the second phase accepts the supposed answer A to the query as correct, then the answer A must be correct. However, if it is not the case that the second phase is working with the true fair split tree, then it may not be possible to guarantee that the certification trail needed to satisfy the second phase will be of constant size.

We summarize this chapter in the following lemma.

Lemma 4.2.2: *k -approximate nearest neighbor queries can be certified in an on-line fashion. When the point set is of size n , the first phase has a worst-case time complexity of $O(\log n)$ per query, and the second phase has a worst-case time complexity of $O(1)$ per query.*

Before the first query is processed, both phases must build a copy of the fair split tree, which takes $O(n \log n)$ time. Alternatively, the first phase could build the fair split tree, and then give the second phase a copy of the tree and also a certification trail. The second phase can in $O(n)$ time verify that the tree it is given is a well formed multi-dimensional search tree, and hence that the structure of tree is suitable to ensure that if the checks for a query succeed, then the supposed answer must be correct.

Chapter 5

Basic Parallel Techniques

In this chapter we present two general techniques for certifying parallel programs, and we also present certification trail solutions to some common problems in parallel computation. The first general technique is a simulation technique, and can be used on any parallel algorithm. Although it gives no reduction of work for the second phase, it can be used for constructing a certification trail for a subroutine of a more complicated algorithm. The second technique involves the parallel evaluation of a sequence of set manipulation operations. We present two applications of this technique. The first is certifying the decision problem of determining if there is an intersection among a set of isothetic line segments. The second is certifying the problem of finding the maximal points of a set of points in three dimensions. In the last section, we show how lowest common ancestor queries, and also range maxima queries can be efficiently certified.

The model of parallel computation that we consider in this chapter is the parallel random-access memory (PRAM) model [50]. In this model, all of the processors are controlled by a common clock, and the computation proceeds in a synchronous manner. Processors have a local memory, and there is also a global memory which is shared by the processors. In the exclusive read exclusive write (EREW) PRAM, no simultaneous accesses are allowed to a single memory location. In the concurrent read exclusive write (CREW) PRAM, simultaneous read operations are allowed on a single memory location. In the concurrent read concurrent write (CRCW) PRAM, both simultaneous reads and writes are allowed to a memory location during a single time step. Here, it is assumed that any read to a memory location take place before the writes. In the CRCW PRAM, some method of resolving which value is written during a concurrent write must be selected. In the common CRCW PRAM, all of the values that are written to a memory location must be the same, and if they are different the machine crashes. In the arbitrary CRCW PRAM, an arbitrary processor wins the write. In the priority CRCW PRAM, the smallest value is written into the memory location. Other methods for resolving write conflicts are possible.

Most of the second phase programs that we present are for a model that we call the polling CREW PRAM, which is slightly more powerful than a CREW PRAM, but is less powerful than a CRCW PRAM. In a polling CREW PRAM, concurrent reads, but not concurrent writes are allowed during the computation. At the last step a one bit global OR operation is performed among the processors to signal an error.

A polling EREW PRAM is defined in an analogous way. Using the polling CREW, we are able to certify problems so that the second phase runs in only constant time. In this chapter, we say that a problem has been optimally certified if the first phase runs with the same processor and time complexity as an optimal algorithm which solves the problem, and the second phase runs in constant time with a linear number of processors on a polling CREW PRAM.

5.1 Simulation Lemma

In this section we present a simulation technique for certifying parallel algorithms. Although the second phase performs the same amount of work as the first phase, it runs in only constant time, and the technique be used for any parallel algorithm. This is a theoretically useful result. Suppose one is trying to certify a parallel algorithm A which computes an answer by calling subroutines B and C , and that B runs with $O(P_B(n))$ processors in $O(T_B(n))$ time and can be optimally certified. Further suppose that C runs with $O(P_C(n))$ processors in $O(T_C(n))$ time, and no efficient certifier is known for C . The results of this section imply that A can be certified so that the first phase has the same complexity as A , and the second phase runs in constant time with $O(n + T_C(n)P_C(n))$ processors, which is optimal when the total work performed by C is $O(n)$.

We now describe the PRAM model that we are using in more detail. We have a set of independent processors, each executing a separate program. We assume a synchronous model where every processor executes one program instruction per time step. Programs contain three types of instructions: branches, conditional branches, and assignment statements. Assignment statements can have a constant number of source operands and one destination operand. Each source operand can be either a constant, a local register, a memory address, or an indirect address. The destination operand can be either a local register, a memory address, or an indirect address. For an indirect address, a local register is specified, and the contents of the register are used as a pointer to a global memory location. Normal arithmetic operations are allowed on the operands. In order to simplify the simulation, processors are assumed to have only a constant number of registers.

Lemma 5.1.1: *Let A be a parallel algorithm in the CREW, EREW, arbitrary, pri-*

ority or common CRCW model. Let A run in $T(n)$ time with $P(n)$ processors, and let the highest memory address used by A be $O(T(n) * P(n))$. Then there exists programs Sim_1^A and Sim_2^A with the following properties:

- Sim_1^A and Sim_2^A certify the problem solved by A .
- Sim_1^A uses the same model as A .
- Sim_1^A generates a certification trail of size $O(T(n) * P(n))$.
- If A runs on a CREW, EREW, arbitrary CRCW or priority CRCW then Sim_1^A runs with asymptotically the same amount of time and processors as A . If A runs on a common CRCW then there is an additive $O(\log n)$ term in the time used.
- Sim_2^A runs in constant time using $O(T(n) * P(n))$ processors.
- If A runs on a common CRCW then Sim_2^A runs on a common CRCW. Otherwise, Sim_2^A runs on a polling CREW.

Proof: We now describe the operation of Sim_1^A . On any given input, we simulate A and leave behind a complete trace of the run. We call this trace the *SimStruct* trace. Sim_1^A is constructed from A so that at every step a *memory record* is created for each memory cell which is written to during the time step, and also a *processor record* is made of the instruction executed by each processor. All of the memory records exist in a one dimensional array, though the memory records for a given memory cell are maintained in a doubly linked list. Each memory record contains the following fields: (i) the memory address for the corresponding memory cell, (ii) the computation step during which the write was performed, (iii) the new value of the memory cell, (iv) an integer representing the processor that performed the write, (v) an index for the previous memory record for the memory cell, and (vi) an index for the next memory record for the memory cell. During the simulation, to aid finding the most recent memory record, memory cells will contain two fields: (i) an index giving the first memory record created for the memory cell, (ii) an index giving the last memory record created for the memory cell. The second field is updated each time a new memory record is created because of a write to the corresponding memory cell. At the start of execution, an initial memory record is created for each memory cell.

Processor records are maintained in a two dimensional array, which is indexed by processor number and computation step. Each processor record contains

Figure 5.1: Illustration of simulation.

Processor Records

Time

Processors	1	2	3	4	5	6	7	8	9
0	PC: 1 M1: - R1: 0 M2: - R2: - M3: -	PC: 2 M1: 0 R1: 0 M2: - R2: 6 M3: -	PC: 3 M1: 4 R1: 0 M2: - R2: 6 M3: -	PC: 4 M1: - R1: 1 M2: - R2: 6 M3: -	PC: 5 M1: 5 R1: 1 M2: - R2: 8 M3: -	PC: 6 M1: 8 R1: 1 M2: - R2: 8 M3: -	PC: 7 M1: - R1: 2 M2: - R2: 8 M3: -	PC: 8 M1: 9 R1: 2 M2: - R2: 11 M3: -	PC: 9 M1: 12 R1: 2 M2: - R2: 11 M3: -
1	PC: 1 M1: - R1: 1 M2: - R2: - M3: -	PC: 2 M1: 1 R1: 1 M2: - R2: 2 M3: -	PC: 3 M1: 5 R1: 1 M2: - R2: 2 M3: -	PC: 4 M1: - R1: 2 M2: - R2: 2 M3: -	PC: 5 M1: 6 R1: 2 M2: - R2: 3 M3: -	PC: 6 M1: 9 R1: 2 M2: - R2: 3 M3: -	PC: 7 M1: - R1: 3 M2: - R2: 3 M3: -	PC: 8 M1: 10 R1: 3 M2: - R2: 7 M3: -	PC: 9 M1: 13 R1: 3 M2: - R2: 7 M3: -
2	PC: 1 M1: - R1: 2 M2: - R2: - M3: -	PC: 2 M1: 2 R1: 2 M2: - R2: 1 M3: -	PC: 3 M1: 6 R1: 2 M2: - R2: 1 M3: -	PC: 4 M1: - R1: 3 M2: - R2: 1 M3: -	PC: 5 M1: 7 R1: 3 M2: - R2: 4 M3: -	PC: 6 M1: 10 R1: 3 M2: - R2: 4 M3: -	PC: 7 M1: - R1: 0 M2: - R2: 4 M3: -	PC: 8 M1: 11 R1: 0 M2: - R2: 13 M3: -	PC: 9 M1: 14 R1: 0 M2: - R2: 13 M3: -
3	PC: 1 M1: - R1: 3 M2: - R2: - M3: -	PC: 2 M1: 3 R1: 3 M2: - R2: 3 M3: -	PC: 3 M1: 7 R1: 3 M2: - R2: 3 M3: -	PC: 4 M1: - R1: 0 M2: - R2: 3 M3: -	PC: 5 M1: 4 R1: 0 M2: - R2: 9 M3: -	PC: 6 M1: 11 R1: 0 M2: - R2: 9 M3: -	PC: 7 M1: - R1: 1 M2: - R2: 9 M3: -	PC: 8 M1: 8 R1: 1 M2: - R2: 17 M3: -	PC: 9 M1: 15 R1: 1 M2: - R2: 17 M3: -

Memory Records

Addr: 0 Rec #: 14 Val: 13 Next: - Time: 9 Prev: 11 Proc: 2	Addr: 1 Rec #: 15 Val: 17 Next: - Time: 9 Prev: 8 Proc: 3	Addr: 2 Rec #: 12 Val: 11 Next: - Time: 9 Prev: 9 Proc: 0	Addr: 3 Rec #: 13 Val: 7 Next: - Time: 9 Prev: 10 Proc: 1
Addr: 0 Rec #: 11 Val: 9 Next: 14 Time: 6 Prev: 4 Proc: 3	Addr: 1 Rec #: 8 Val: 8 Next: 15 Time: 6 Prev: 5 Proc: 0	Addr: 2 Rec #: 9 Val: 3 Next: 12 Time: 6 Prev: 6 Proc: 1	Addr: 3 Rec #: 10 Val: 4 Next: 13 Time: 6 Prev: 7 Proc: 2
Addr: 0 Rec #: 4 Val: 6 Next: 11 Time: 3 Prev: 0 Proc: 0	Addr: 1 Rec #: 5 Val: 2 Next: 8 Time: 3 Prev: 1 Proc: 1	Addr: 2 Rec #: 6 Val: 1 Next: 9 Time: 3 Prev: 2 Proc: 2	Addr: 3 Rec #: 7 Val: 3 Next: 10 Time: 3 Prev: 3 Proc: 3
Addr: 0 Rec #: 0 Val: 5 Next: 4 Time: 0 Prev: - Proc: -	Addr: 1 Rec #: 1 Val: 1 Next: 5 Time: 0 Prev: - Proc: -	Addr: 2 Rec #: 2 Val: 0 Next: 6 Time: 0 Prev: - Proc: -	Addr: 3 Rec #: 3 Val: 2 Next: 7 Time: 0 Prev: - Proc: -

Memory Cells

First: 0 Last: 14	First: 1 Last: 15	First: 2 Last: 12	First: 3 Last: 13
0	1	2	3

Program

- 1: R1 = ProcID()
- 2: R2 = *R1 + 1
- 3: *R1 = R2
- 4: R1 = (R1 + 1) mod 4
- 5: R2 = R2 + *R1
- 6: *R1 = R2
- 7: R1 = (R1 + 1) mod 4
- 8: R2 = R2 + *R1
- 9: *R1 = R2

several fields: (i) the program counter (that is, a pointer to the instruction that the processor just executed), (ii) for each memory cell read, the index of the memory record that was accessed, and (iii) for the memory cell written to, an index of the memory record that was created for the write. If no memory locations are either read from or written to during an instruction these fields are left as *nil*. In the CREW and EREW models each processor is allocated a blank memory record at the beginning of each computation step. If a write is performed the processor assigns the fields of the unused memory record and adds it onto the end of the doubly linked list for that memory cell. In the arbitrary and priority CRCW models each processor is again allocated a blank memory record at the beginning of each computation step. If during a step multiple processors write to a memory cell, one of the processors is elected as the winner of the write according to the rules of the model, and this processor assigns the fields of the memory record it was allocated and sets up the appropriate links.

See Figure 5.1 for an example of this simulation. The operation $x \bmod y$ is the integer remainder when x is divided by y . Each processor has two registers R_1 and R_2 , and we denote a pointer dereference by $*R_1$ or $*R_2$. A “—” in a field means that the value of the field is *nil* or is undefined. The fields M_1 , M_2 , and M_3 are used for storing the indices of the memory records accessed. Since at most one memory record is referenced during each operation, the fields M_2 and M_3 have value *nil*. All of the processors execute the same program, which is given at the bottom of the figure.

The only difficulty is in the common CRCW model. The problem is that there is no obvious method of electing one processor as the winner of the write to a memory cell. To get around this problem, we change memory records to have only two fields: one for the computation step, and another for the new value of the memory cell. Now we simulate the program twice with the first run counting the number of memory records each memory cell needs. Before the second run each memory cell is allocated enough memory records to record every change to the memory cell. In this case all of the memory records for a cell are located in a contiguous block instead of a doubly linked list. Now, all the processors writing to a memory cell concurrently assign the fields of the next memory record.

We now describe the operation of Sim_2^A . To verify the computation several checks need to be performed on the *SimStruct* trace. The initial value of each memory cell as stored in the first memory record for the memory cell must be correct. The

time values in the doubly linked list of memory records for a memory cell must be monotonically increasing. Also, each processor record is checked to see if: (i) the program counter is consistent with the program counter in the processor record for the previous step, (ii) the registers are consistent with the registers in the processor record for the previous step, (iii) the pointers to the memory records read from or written to have the appropriate time fields, and (iv) the value of the memory record or the register that is written to is correct. In the priority CRCW model the other processors which attempted to write to the cell check that the processor which won had the highest priority write. In the CREW, EREW, priority and arbitrary CRCW models, processors assigned to each memory record check that there is a processor record which is responsible for creating the memory record. In the common CRCW model, each memory record is also given an initially zero one bit field. The processor assigned to the processor record writes a one to the memory record that the processor record attempted to write to during the computation. Now if every memory record has a one, there are no extraneous entries in the memory records for all of the memory cells.

If A runs on a CREW, EREW, arbitrary or priority CRCW model the initial simulation can be done in the same model without any asymptotic penalty in the time, processors or space used. If A runs on the common CRCW model then the simulation is performed on a common CRCW model, and has an additive $O(\log n)$ time penalty which is needed for the initial memory record allocation. If A runs on a CREW, EREW, arbitrary or priority CRCW, then $Sim_{\frac{A}{2}}^A$ runs in constant time on a polling CREW. If A runs on a common CRCW, then $Sim_{\frac{A}{2}}^A$ runs in constant time on a common CRCW.

There are two parts in proving that the simulation to construct the *SimStruct* trace and the checks on the *SimStruct* trace have the desired certification properties. First we note that if there are no faults during the simulation, then the checks on the *SimStruct* trace will succeed during the run of $Sim_{\frac{A}{2}}^A$. The more difficult part is showing that if the checks on the *SimStruct* trace succeed then the output is correct.

Consider the execution of A . We introduce some useful notation. Let $MEM_i(t)$ be the value of the i th memory cell at the start of the t th computation step. Let $PC^p(t)$ be the value of the program counter for the p th processor at the start of the t th computation step. Finally, let $R_i^p(t)$ be the value of the i th register

in the p th processor at the start of the t th computation step.

Given a *SimStruct* trace that has passed the checks performed by Sim_2^A , we present some more notation. Let $\overline{PC}^p(t)$ be the value of the program counter in the processor record indexed by p and t . Let $\overline{R}_i^p(t)$ be the value of the i th register in the processor record indexed by p and t . Finally, let $\overline{MEM}_i(t)$ be the value in the memory record for the i th memory cell at time t . The memory record for the i th memory cell at time t is the memory record for memory cell i with the largest time stamp that is less than t .

It can be shown by induction on the time step that if the *SimStruct* trace has passed the checks performed by Sim_2^A , then $PC^p(t) = \overline{PC}^p(t)$, $R_i^p(t) = \overline{R}_i^p(t)$, and $MEM_i(t) = \overline{MEM}_i(t)$ for all values of p , t and i . We omit the details. ■

5.2 Parallel Sequence Evaluation Structure

In this section we present algorithms for certifying the parallel evaluation of certain set manipulation operations. The basic operand type is again pairs of the form (it, val) . it is a positive integer, and is called the item number of the pair. Item numbers are used for referring to pairs, and hence all pairs must have distinct item numbers. val can be an arbitrary real number, and is called the item value of the pair. We write $(it, val) < (it', val')$ iff $val < val'$ or $(val = val'$ and $it < it')$. For convenience, we restate the definitions of the operations that we consider in this chapter.

- *insert*(it, val): Insert the pair (it, val) into the set. The item number it must not have been used previously.
- *delete*(it): Delete the pair with item number it from the set. Some pair with item number it must be in the set.
- *successor*(it, val): Return the smallest pair that is larger than (it, val) that is contained in the set. If there is no pair larger than (it, val) in the set, then return the value *nil*.
- *predecessor*(it, val): Return the largest pair that is smaller than (it, val) that is contained in the set. If there is no pair smaller than (it, val) in the set, then return the value *nil*.

- *nullop*: The null operation. The set is not modified, nor is a value returned.

We sometimes abbreviate *predecessor* as *pred* and *successor* as *succ*. Given a sequence of the above operations, we define $pred_i(it, val)$ to be the predecessor of (it, val) in the set immediately after the i th operation has been performed. Similarly, we define $succ_i(it, val)$ to be the successor of (it, val) in the set immediately after the i th operation has been performed.

We now present some definitions in a Pascal like style for the Parallel Sequence Evaluation (PSE) structure, which is useful for certifying the parallel sequence evaluation problem. The PSE structure is defined for a sequence of *insert*, *delete*, and *nullop* operations. For the definitions below, let *MAXID* be the maximum item number that appears in the sequence. We assume that the first two operations in the sequence are $insert(MAXID - 1, -\infty)$ and $insert(MAXID, \infty)$. If we are given a sequence for which this does not hold, we increase the value of *MAXID* by two and augment the sequence with these two operations.

TYPE

```
PredRecord =
  RECORD
    Start : integer;
    Finish : integer;
    PredID : integer;
    PredIndex : integer;
  END;
```

```
SuccRecord =
  RECORD
    Start : integer;
    Finish : integer;
    SuccID : integer;
    SuccIndex : integer;
  END;
```

```
ElementRecord =
  RECORD
    Value : value_type;
```

```

    InsTime : integer;
    DelTime : integer;
    SizePred : integer;
    Preds : array [1..SizePred] of PredRecord;
    SizeSucc : integer;
    Succs : array [1..SizeSucc] of SuccRecord;
END;
```

```
PSEtype = array [1..MAXID] of ElementRecord;
```

Assume that we are given a sequence of n operations, and that we have allocated the structure Θ of type *PSEtype*. Consider a pair (it, val) appearing in the sequence. We now describe what the fields of the record $\Theta[it]$ contain. First, $Value = val$ holds.

Let i be the number of the operation which inserts the pair, and let d be the number of the operation which deletes the pair (if (it, val) is never deleted then let $d = n + 1$). Then $InsTime = i$ and $DelTime = d$ will both hold.

Now we describe the contents of the field *SizePred* and the array *Preds*. First, for the pair $(MAXID - 1, -\infty)$, we have $SizePred = 1$, and *Preds* will contain the single record with $Start = 1$, $Finish = n + 1$, $PredID = nil$, and $PredIndex = nil$.

So assume $(it, val) \neq (MAXID - 1, -\infty)$. Let

$$T = \{t \mid (t = i) \text{ or } (t = d) \text{ or } ((pred_t(it, val) \neq pred_{t-1}(it, val)) \text{ and } (i < t < d))\}.$$

Then $SizePred = |T| - 1$ will hold. Let $T' = (t_1, t_2, \dots, t_{|T|})$ be the indices in T in sorted order. Then for $1 \leq j < |T|$, the following hold:

1. $Preds[j].Start = t_j$.
2. $Preds[j].Finish = t_{j+1} - 1$.

Let $(it', val') = pred_{t_j}(it, val)$.

3. $Preds[j].PredID = it'$
4. $Preds[j].PredIndex$ is the index of the successor record for (it', val') for time t_j .

The contents of the field *SizeSucc* and the array *Succs* are similar. First, for the pair $(MAXID, \infty)$, $SizeSucc = 1$, and *Succs* will contain the single record with

$Start = 2,$

$Finish = n + 1,$ $SuccID = nil,$ and $SuccIndex = nil.$

So assume $(it, val) \neq (MAXID, \infty).$ Let

$$T = \{t \mid (t = i) \text{ or } (t = d) \text{ or } ((succ_t(it, val) \neq succ_{t-1}(it, val)) \text{ and } (i < t < d))\}.$$

Then $SizeSucc = |T| - 1$ will hold. Let $T' = (t_1, t_2, \dots, t_{|T|})$ be the indices in T sorted. Then for $1 \leq j < |T|,$ the following hold:

1. $Succs[j].Start = t_j.$

2. $Succs[j].Finish = t_{j+1} - 1.$

Let $(it', val') = succ_{t_j}(it, val).$

3. $Succs[j].SuccID = it'.$

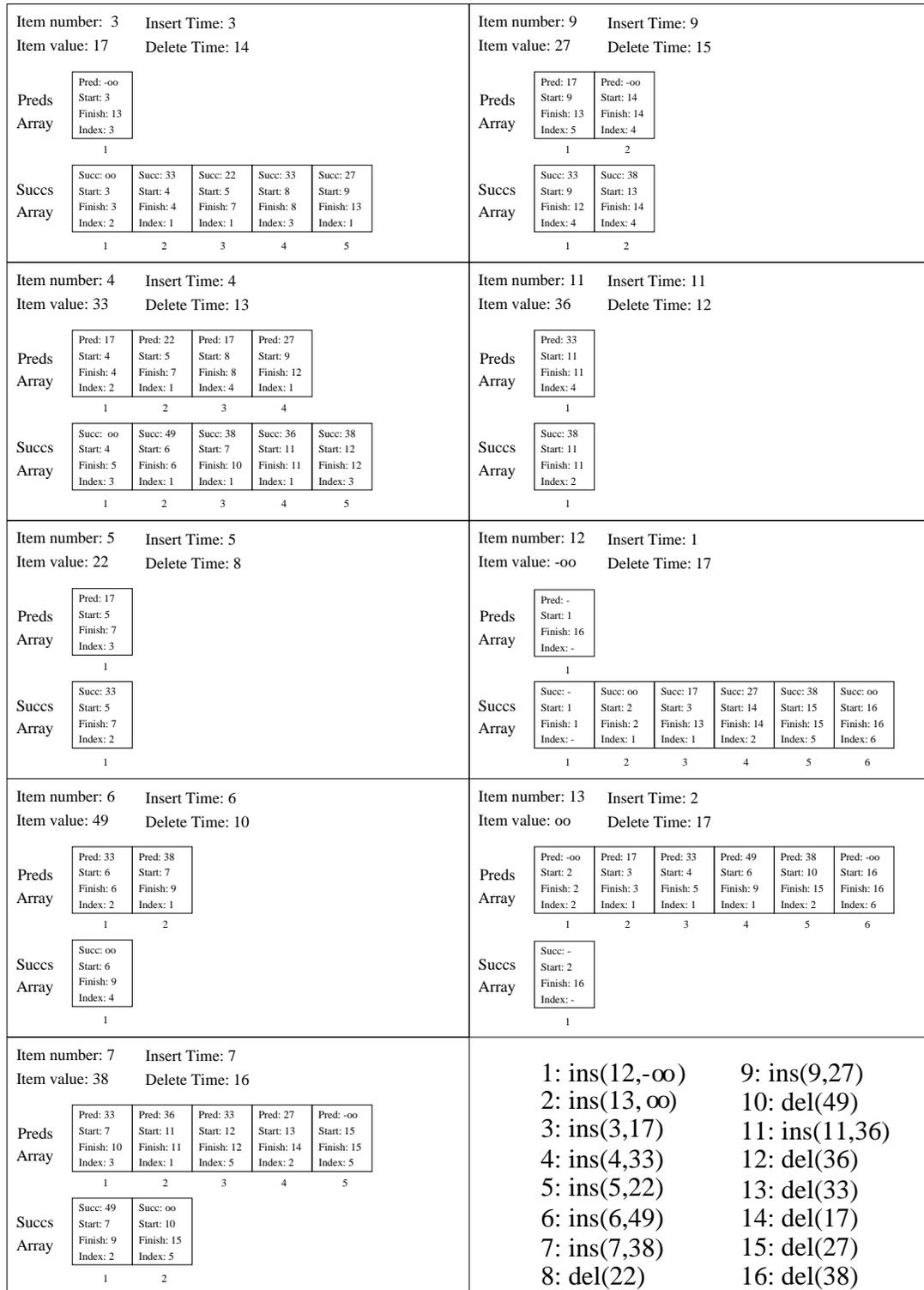
4. $Succs[j].SuccIndex$ is the index of the predecessor record for (it', val') for time $t_j.$

The index of the predecessor record for a pair (it, val) at time i is the index $j,$ such that $1 \leq j \leq \Theta[it].SizePred$ and $\Theta[it].Preds[j].Start \leq i \leq \Theta[it].Preds[j].Finish.$ In a similar manner we define the index of the successor record for a pair at a specific time. Since $succ_1(MAXID - 1, -\infty) = nil,$ for the specific record $\Theta[MAXID - 1].Succs[1],$ we set $SuccID = nil$ and $SuccIndex = nil.$

See Figure 5.2 for an example. To simplify the figure, each *delete* operation is given the item value instead of the item number of the pair being deleted. Also, in all of the *Preds* and *Succs* arrays, instead of giving the item numbers of the predecessor and successor pairs, we give the item values instead. As an example of how the PSE structure for a sequence can be used, consider the third element in the *Succs* array for the pair $(3, 17).$ Since the value for *Start* is 5, and the value for *Finish* is 7, we see that the pair $(5, 22)$ is the immediate successor of the pair $(3, 17)$ immediately after the 5th operation is performed, immediately after the 6th operation is performed, and also immediately after the 7th operation is performed.

Given a sequence of *insert*, *delete*, and *nullop* operations, the PSE structure can be used to help check the correctness of *successor* and *predecessor* queries occurring at different times in the sequence. Assume a first phase program finds (it', val') as the answer to $pred_t(it, val).$ The first phase can perform a binary search on the

Figure 5.2: Illustration of Parallel Sequence Evaluation structure



array $\Theta[it'].Succs$ to find the index i of the record corresponding to time t . The second phase can now be given the query $pred_t(it, val)$, the supposed answer (it', val') , and i as the certification trail. The second phase performs the following checks: (i) that $\Theta[it'].Value = val'$, (ii) that $(it', val') < (it, val)$, (iii) that $\Theta[it'].Succs[i]$ is the successor record for (it', val') at time t , and (iv) that $\Theta[it'].Succs[i].SuccID$ is the item number of a pair that is larger than or equal to (it, val) . If these four checks succeed, then it must be the case that (it', val') is the correct answer to the query.

Note that if for the query $pred_t(it, val)$, the pair (it, val) is present in the set at time t , then the answer to the query can be found by a binary search on the array $\Theta[it].Preds$ to find the predecessor record for time t . Similarly, if the operand to a *successor* query is present in the set, a binary search on the successor records for the pair can be performed to determine the answer to the query.

5.2.1 Constructing and Checking the PSE Structure

In this section we present the algorithm *MakePSE* for constructing the Parallel Sequence Evaluation structure, and the algorithm *CheckPSE* for checking the correctness of the Parallel Sequence Evaluation structure. Note that *CheckPSE* is a program checker, and does not need a certification trail to run efficiently. When we say a condition holds at time i , we mean that the condition holds immediately after the i th operation is performed.

Algorithm *MakePSE*: We are given a sequence of n operations.

1. Create two arrays, *PredArray* and *SuccArray*, each capable of holding $3n$ quadruples.
2. Consider the i th operation. If it is of the form $insert(it, val)$ perform the following steps:
 - (a) Create a *PredArray* record $((it, val), i, pred_i(it, val), \text{"start"})$ and a *SuccArray* record $((it, val), i, succ_i(it, val), \text{"start"})$.
 - (b) Create a *SuccArray* record $(pred_i(it, val), i, (it, val), \text{"start"})$, and a *PredArray* record $(succ_i(it, val), i, (it, val), \text{"start"})$.
 - (c) Create a *PredArray* record $(succ_i(it, val), i - 1, pred_i(it, val), \text{"finish"})$, and a *SuccArray* record $(pred_i(it, val), i - 1, succ_i(it, val), \text{"finish"})$.
3. Consider the i th operation. If it is of the form $delete(it)$, let val be the value of the pair with item number it , and perform the following steps:

- (a) Create a *PredArray* record $((it, val), i-1, pred_i(it, val), \text{"finish"})$ and a *SuccArray* record $((it, val), i-1, succ_i(it, val), \text{"finish"})$.
 - (b) Create a *PredArray* record $(succ_i(it, val), i-1, (it, val), \text{"finish"})$ and a *SuccArray* record $(pred_i(it, val), i-1, (it, val), \text{"finish"})$.
 - (c) Create a *PredArray* record $(succ_i(it, val), i, pred_i(it, val), \text{"start"})$ and a *SuccArray* record $(pred_i(it, val), i, succ_i(it, val), \text{"start"})$.
4. For every pair (it, val) that is inserted but never deleted, perform step 3 on the pair with the value of i set to $n+1$.
 5. Sort all of the *PredArray* records lexicographically (with $\text{"start"} < \text{"finish"}$), and then remove any duplicate entries.
 6. Sort all of the *SuccArray* records lexicographically (with $\text{"start"} < \text{"finish"}$), and then remove any duplicate entries.

All of the predecessor records for a pair are stored in a contiguous portion of *PredArray*. Similarly, all of the successor records for a pair are stored in a contiguous portion of *SuccArray*.

7. Use *PredArray* and *SuccArray* to construct the PSE structure.

The correctness of *MakePSE* follows from the next lemma.

Lemma 5.2.1: *After step 6 of MakePSE is executed, the following two statements hold:*

1. $(P_1, t, P_2, \text{"start"})$ is an entry in *SuccArray* (*PredArray*), iff P_2 is the successor (predecessor) of P_1 at time t but not $t-1$.
2. $(P_1, t, P_2, \text{"finish"})$ is an entry in *SuccArray* (*PredArray*), iff P_2 is the successor (predecessor) of P_1 at time t but not $t+1$.

Proof: We will only prove that $(P_1, t, P_2, \text{"start"})$ is an entry in *SuccArray*, iff P_2 is the successor of P_1 at time t but not $t-1$. The rest of the proof uses similar techniques and is omitted.

Assume that $(P_1, t, P_2, \text{"start"})$ is an entry in *SuccArray*. The first case is that it was generated by a processor assigned to an *insert* operation. Clearly P_2 must be the successor of P_1 at time t . Also, since one of P_1 or P_2 was just added to the set, P_2 was not the successor of P_1 at time $t-1$. The second case is that it was generated by a processor assigned to a *delete* operation. Clearly, P_2 must be the successor of P_1

at time t . We see that the operand to the *delete* operation must have been a pair P_3 which was in the set at time $t - 1$, and that $P_1 < P_3 < P_2$ must hold. So clearly P_2 was not the successor of P_1 at time $t - 1$.

Now assume that P_2 is the successor of P_1 at time t but not at time $t - 1$. The first case is that either P_1 or P_2 was inserted into the set at time t . Then the *SuccArray* record $(P_1, t, P_2, \text{"start"})$ will be in *SuccArray*. The second case is that neither P_1 or P_2 were inserted into the set at time t . Then at time $t - 1$ there was a pair $P_3 = (it, val)$ in the set such that $P_1 < P_3 < P_2$. But since P_3 is not in the set at time t (else P_2 would not be the successor of P_1 at time t), the $(t - 1)$ th operation in the sequence must be *delete(it)*. That implies that the record $(P_1, t, P_2, \text{"start"})$ will be in *SuccArray*. ■

Lemma 5.2.2: *Given a sequence of insert, delete, and nulloperations of length n , MakePSE constructs the PSE structure of the sequence, and runs in $O(\log n)$ time with $O(n)$ processors on a CREW PRAM.*

Proof: The sorting for steps five and six can be performed using the parallel sorting algorithm of [?]. The only difficulty in the implementation is finding the values of $pred_i(it, val)$ and $succ_i(it, val)$ in steps two and three of *MakePSE*. These values can be found using the techniques of [5]. In this paper, the authors show how certain sequences of set manipulation operations can be efficiently evaluated in parallel. For the specific case of *insert*, *delete*, *predecessor*, and *successor* operations, the authors show how such a sequence can be evaluated in $O(\log n)$ time with $O(n)$ processors on a CREW PRAM. It is a simple matter to use their techniques for our problem. ■

We now consider the problem of checking the PSE structure.

Algorithm CheckPSE: We are given a sequence of n operations and the structure Θ .

1. For each $1 \leq i \leq n$, perform the appropriate check:
 - (a) If the i th operation is of the form *insert(it, val)*, check that $\Theta[it].InsTime = i$, $\Theta[it].Value = val$, and $1 \leq \Theta[it].DelTime \leq n + 1$.
 - (b) If the i th operation is of the form *delete(it)*, check that $\Theta[it].DelTime = i$.
2. For each (it, val) such that $(it, val) \neq (MAXID - 1, -\infty)$ and $(it, val) \neq (MAXID, \infty)$ perform the following checks on $\Theta[it].SizePred$ and $\Theta[it].Preds$:

- (a) $\Theta[it].Preds[1].Start = \Theta[it].InsTime.$
 - (b) $\Theta[it].Preds[\Theta[it].SizePred].Finish = \Theta[it].DelTime - 1.$
 - (c) For each $1 \leq j \leq \Theta[it].SizePred$ perform the following check:
 - i. $\Theta[it].Preds[j].Start \leq \Theta[it].Preds[j].Finish.$
 - (d) For each $1 \leq j < \Theta[it].SizePred$ perform the following checks:
 - i. $\Theta[it].Preds[j].Finish = \Theta[it].Preds[j + 1].Start - 1.$
 - ii. $\Theta[it].Preds[j].PredID \neq \Theta[it].Preds[j + 1].PredID.$
 - (e) For each $1 \leq j \leq \Theta[it].SizePred$ perform the following checks:

Let $PredID = \Theta[it].Preds[j].PredID, PredIndex = \Theta[it].Preds[j].PredIndex.$

 - i. $1 \leq PredID \leq MAXID.$
 - ii. $1 \leq PredIndex \leq \Theta[PredID].SuccSize.$
 - iii. $\Theta[it].Preds[j].Start = \Theta[PredID].Succs[PredIndex].Start.$
 - iv. $\Theta[it].Preds[j].Finish = \Theta[PredID].Succs[PredIndex].Finish.$
 - v. $\Theta[PredID].Succs[PredIndex].SuccID = it.$
3. For each (it, val) such that $(it, val) \neq (MAXID - 1, -\infty)$ and $(it, val) \neq (MAXID, \infty)$ perform analogous checks on $\Theta[it].SizeSucc$ and $\Theta[it].Succs.$
4. Perform the following checks for $(MAXID - 1, -\infty)$:
- (a) $\Theta[MAXID - 1].SizePred = 1.$
 - (b) The record $\Theta[MAXID - 1].Preds[1]$ has fields $Start = 1, Finish = n + 1, PredID = nil,$ and $PredIndex = nil.$
 - (c) The record $\Theta[MAXID - 1].Succs[1]$ has fields $Start = 1, Finish = 1, PredID = nil,$ and $PredIndex = nil.$
 - (d) $(MAXID - 1, -\infty)$ passes the checks in step 2 (except for check (e) on the first record of $\Theta[MAXID - 1].Succs).$
5. Perform the following checks for $(MAXID, \infty)$:
- (a) $\Theta[MAXID - 1].SizeSucc = 1.$
 - (b) The record $\Theta[MAXID].Succs[1]$ has fields $Start = 2, Finish = n + 1, SuccID = nil,$ and $SuccIndex = nil.$
 - (c) $(MAXID, \infty)$ passes the checks in step 3.
6. If all the checks succeed output *ok*, else output *error*.

Lemma 5.2.3: *Given a sequence of insert, delete, and nulloperations of length n , and the supposed PSE structure Θ of the sequence, *CheckPSE* checks the correctness of Θ and runs in constant time with $O(n)$ processors on a polling CREW PRAM.*

Proof: First we note that if Θ is correct, then *CheckPSE* will output *ok*.

Now, by way of contradiction assume that *CheckPSE* outputs *ok* but Θ is not correct. Because of all of the checks performed, most ways that Θ could fail to be the PSE structure are ruled out. Specifically, the *Preds* and *Succs* arrays for all of the pairs must be well formed, meaning there is a unique predecessor and successor record for each pair for each time step for which the pair is contained in the set, and all of these records must appear in sorted order. One possible way that Θ could fail to be correct is that there is a pair N active at some time t such that the predecessor record for N at time t indicates M as the predecessor of N , but P is really the predecessor of N at time t . It is easy to show that N , M , and P must all be active at time t . Also, $M < P < N$ must hold. For the pair N , there is a sequence A and positive integer x , such that $a_1 = N$ and for all $1 \leq j < x$ we have that the predecessor record in Θ for a_j at time t indicates a_{j+1} as the predecessor. A must be finite, since $a_j > a_{j+1}$ for all $1 \leq j < x$, and there are only a finite number of pairs in the sequence. Thus we can write $A = (a_1, a_2, \dots, a_x)$ for some x . Also, we can show that $a_x = (MAXID - 1, -\infty)$. By our hypothesis, it follows that $a_2 = M$. So now consider the sequence $B = (b_1, b_2, \dots, b_y)$ implied by P . The checks in 2(e) ensure that a pair can be given as the predecessor of at most one other pair for any time t . Note that $b_y = (MAXID - 1, -\infty) = a_x$. Since $P < N$ and $P \neq M$, we see that $B = (a_k, a_{k+1}, \dots, a_x)$ for some value of k greater than or equal to three. But this contradicts our original hypothesis that $P > M$.

The other possible way that Θ could fail to be correct is that there is a pair N in Θ active at some time t such the successor record for N at time t indicates M as the successor of N at time t , but P is really the successor of N at time t . The proof that this is impossible is similar to the one above, and is omitted. ■

We now present algorithms *EvalSeq₁* and *EvalSeq₂* which certify the parallel sequence evaluation problem.

Algorithm *EvalSeq₁*: we are given as input a sequence Σ of *insert*, *delete*, *predecessor*, *successor*, and *nulloperations*.

1. Compute the answers A to the sequence Σ .

2. Construct Σ' from Σ by replacing the *predecessor* and *successor* operations with *nullop*.
3. Compute $MakePSE(\Sigma') = \Theta$.
4. Construct T as follows. Perform the appropriate step according to the i th operation in Σ :
 - (a) *pred(it, val)*: Let $A[i] = (it', val')$. Let j be the index of the successor record for (it', val') at time i . Set $T[i] = j$.
 - (b) *succ(it, val)*: Let $A[i] = (it', val')$. Let j be the index of the predecessor record for (it', val') at time i . Set $T[i] = j$.
 - (c) *insert(it, val), delete(it), nullop*: Set $T[i] = nil$.
5. Output $(A, (\Theta, T))$.

Algorithm EvalSeq₂: we are given as input $(\Sigma, \tilde{A}, (\tilde{\Theta}, \tilde{T}))$.

1. Construct Σ' as above.
2. Check that $CheckPSE(\Sigma', \tilde{\Theta}) = ok$.
3. For each $1 \leq i \leq n$ perform the appropriate checks according to the i th operation:
 - (a) *pred(it, val)*: Let $\tilde{A}[i] = (it', val')$. Check that:
 - i. $1 \leq it' \leq MAXID$.
 - ii. $\tilde{\Theta}[it'].Value = val'$.
 - iii. $(it', val') < (it, val)$.
 - iv. $1 \leq \tilde{T}[i] \leq \tilde{\Theta}[it'].SuccSize$.
 - v. $\tilde{\Theta}[it'].Succs[\tilde{T}[i]].Start \leq i \leq \tilde{\Theta}[it'].Succs[\tilde{T}[i]].Finish$.
 - vi. The pair with item number $\tilde{\Theta}[it'].Succs[\tilde{T}[i]].SuccID$ is larger than or equal to (it, val) .
 - (b) *succ(it, val)*: Let $\tilde{A}[i] = (it', val')$. Check that:
 - i. $1 \leq it' \leq MAXID$.
 - ii. $\tilde{\Theta}[it'].Value = val'$.
 - iii. $(it', val') > (it, val)$.
 - iv. $1 \leq \tilde{T}[i] \leq \tilde{\Theta}[it'].PredSize$.
 - v. $\tilde{\Theta}[it'].Preds[\tilde{T}[i]].Start \leq i \leq \tilde{\Theta}[it'].Preds[\tilde{T}[i]].Finish$.

- vi. The pair with item number $\tilde{\Theta}[it'].Preds[\tilde{T}[i]].PredID$ is smaller than or equal to (it, val) .

Lemma 5.2.4: *Algorithms $EvalSeq_1$ and $EvalSeq_2$ certify the parallel sequence evaluation problem, and have the following properties: (i) $EvalSeq_1$ runs in $O(\log n)$ time with $O(n)$ processors on a CREW PRAM, (ii) $EvalSeq_2$ runs in $O(1)$ time with $O(n)$ processors on a polling CREW PRAM.*

Proof: Using the results of [5], we can implement the algorithms to have the desired complexity. The correctness follows from the properties of the PSE structure that were discussed earlier. ■

5.2.2 Certifying Intersections of Isothetic Line Segments

We now consider the decision problem of determining whether there are any intersections between a set of horizontal line segments and a set of vertical line segments. All line segments are considered to be “open,” hence only intersections which do not involve endpoints of the line segments are considered. We will present algorithms $IsoIntersect_1$ and $IsoIntersect_2$ which optimally certify this problem.

The following lemma proves to be useful.

Lemma 5.2.5: *It is possible to define programs $Sort_1$ and $Sort_2$ that certify the problem of sorting and have the following properties: (i) $Sort_1$ runs in $O(\log n)$ time with $O(n)$ processors on an EREW PRAM, (ii) $Sort_2$ runs in $O(1)$ time with $O(n)$ processors on a polling CREW PRAM.*

This is a straightforward adaptation of the sequential certifier for the problem of sorting (see the last chapter of this thesis for a description of the sequential certifier for sorting).

Let H be a set of horizontal line segments, and let V be a set of vertical line segments. For all $h \in H$ we define $xleft(h)$ to be the left x coordinate of h , $xright(h)$ to be the right x coordinate of h , and $ycor(h)$ to be the y coordinate of h . For all $v \in V$ we define $ytop(v)$ to be the top y coordinate of v , $ybottom(v)$ to be the bottom y coordinate of v , and $xcor(v)$ to be the x coordinate of v . Let $P_H()$ be any permutation from H to the integers $[1, 2, \dots, |H|]$. Similarly, let $P_V()$ be any

permutation from V to the integers $[1, 2, \dots, |V|]$. Let $P_H^{-1}()$ and $P_V^{-1}()$ be the inverses of $P_H()$ and $P_V()$ respectively.

Algorithm $IsoIntersect_1$: we are given (H, V) as input.

1. Construct the set $X = \{(xleft(h), \text{"ins"}, P_H(h)) \mid h \in H\} \cup \{(xright(h), \text{"del"}, P_H(h)) \mid h \in H\} \cup \{(xcor(v), \text{"query"}, P_V(v)) \mid v \in V\}$.
2. Compute $Sort_1(X) = (O, O_\square)$ sorting the triples lexicographically (with $\text{"del"} < \text{"query"} < \text{"ins"}$).
3. Construct O' from O as follows:
 - (a) If $O[i] = (xright(h), \text{"del"}, P_H(h))$ then set $O'[i] = delete(P_H(h))$.
 - (b) If $O[i] = (xcor(v), \text{"query"}, P_V(v))$ then set $O'[i] = succ(\infty, ybottom(v))$.
 - (c) If $O[i] = (xleft(h), \text{"ins"}, P_H(h))$ then set $O'[i] = insert(P_H(h), ycor(h))$.
4. Compute $EvalSeq_1(O') = (A, A_\square)$.
5. For all i such that $O'[i] = succ(\infty, ybottom(v))$ check that $(-\infty, ytop(v)) < A[i]$.
 - (a) If at least one of these checks fail, let i be an index where they fail and let $A[i] = (P_H(h), ycor(h))$ and $O'[i] = succ(\infty, ybottom(v))$. Then output $(\text{"inter"}, (P_H(h), P_V(v)))$.
 - (b) If all the checks succeed, then output $(\text{"nointer"}, (O, O_\square, A, A_\square))$.

Algorithm $IsoIntersect_2$: we are given $((H, V), ans, \tilde{C})$ as input.

1. If $ans = \text{"inter"}$, check that \tilde{C} is of the form (\tilde{j}, \tilde{k}) with $P_H^{-1}(\tilde{j}) \in H$ and $P_V^{-1}(\tilde{k}) \in V$, and that $P_H^{-1}(\tilde{j})$ and $P_V^{-1}(\tilde{k})$ intersect. If they do, output ok , else output $error$.
2. Else, check that $ans = \text{"nointer"}$. Let \tilde{C} be of the form $(\tilde{O}, \tilde{O}_\square, \tilde{A}, \tilde{A}_\square)$ and continue.
3. Construct the set X as above.
4. Check that $Sort_2(X, \tilde{O}, \tilde{O}_\square) = ok$.
5. Construct O' as above.
6. Check that $EvalSeq_2(O', \tilde{A}, \tilde{A}_\square) = ok$.
7. For all i such that $O'[i] = succ(\infty, ybottom(v))$ check that $(-\infty, ytop(v)) < \tilde{A}[i]$.

8. If all the checks succeed output *ok*, else output *error*.

Lemma 5.2.6: *IsoIntersect₁* and *IsoIntersect₂* certify the problem of testing whether there are any intersections between a set of horizontal line segments and a set of vertical line segments. *IsoIntersect₁* and *IsoIntersect₂* have the following properties: (i) *IsoIntersect₁* runs in $O(\log n)$ time with $O(n)$ processors on a CREW PRAM, (ii) *IsoIntersect₂* runs in $O(1)$ time with $O(n)$ processors on a polling CREW PRAM.

Proof: The correctness of *IsoIntersect₁* and *IsoIntersect₂* follows from the following observation. During the processing of a query $\text{succ}(\infty, y_{\text{bottom}}(v))$, the contents of the set correspond precisely to the elements of

$$H' = \{h \mid h \in H \text{ and } x_{\text{left}}(h) < x_{\text{cor}}(v) < x_{\text{right}}(h)\}$$

Thus, if the answer to the query $\text{succ}(\infty, y_{\text{bottom}}(v))$ is $(P_H(h), y_{\text{cor}}(h))$, then $y_{\text{top}}(v) > y_{\text{cor}}(h)$ iff v intersects h .

The complexity follows from lemma 5.2.4. ■

It is not difficult to modify *IsoIntersect₁* and *IsoIntersect₂* to allow for intersections involving endpoints of the line segments. In step two of *IsoIntersect₁* and step four of *IsoIntersect₂*, we sort X lexicographically with “*ins*” < “*query*” < “*del*”. Also, we interchange the occurrences of $-\infty$ with ∞ , and vice versa. It is easy to show that the resulting algorithms will detect all intersections.

5.2.3 Certifying 3-Dimensional Maxima

In this section we consider the problem of certifying the 3d maxima problem, which we now define. We are given a set P of points in 3-dimensional Euclidean space. We assume that the points are in general position, so that all of the x, y and z coordinates are distinct. Also, we assume that the x and y coordinates of all of the points are greater than zero. We say that a point $p_1 = (x_1, y_1, z_1) \in P$ *dominates* a point $p_2 = (x_2, y_2, z_2) \in P$ if and only if $x_1 \geq x_2$, $y_1 \geq y_2$, and $z_1 \geq z_2$. We wish to output the set of points in P which are not dominated by any other point. Atallah, Cole, and Goodrich present this as an application of cascading mergesort [4]. For every point $p_1 = (x_1, y_1, z_1) \in P$, they find the point $p_2 = (x_2, y_2, z_2) \in P$ with maximal z coordinate such that $x_2 \geq x_1$ and $y_2 \geq y_1$. For any point p_1 , let $\text{maxtwod}(p_1)$

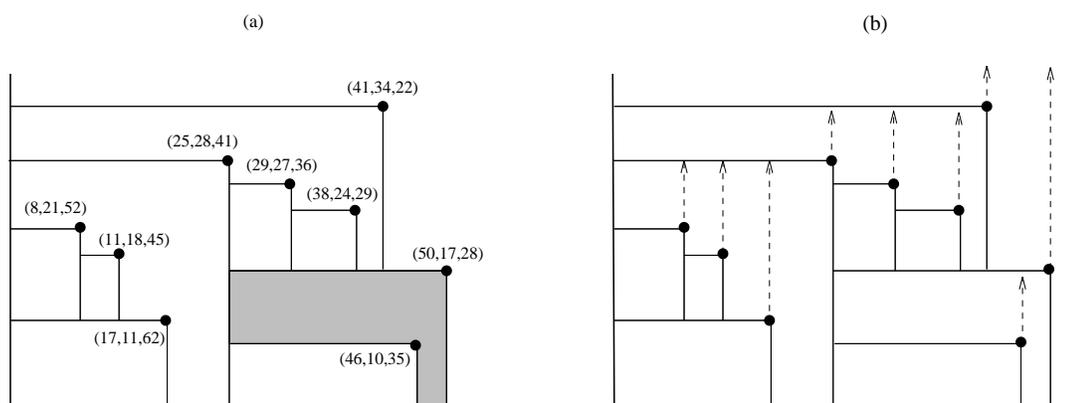


Figure 5.3: Decomposition of XY plane.

denote the point p_2 with this property. If $z_2 \geq z_1$ then p_1 is dominated by point p_2 . Otherwise, if $z_2 < z_1$ then p is not dominated by any point in P . Their algorithm runs in $O(\log n)$ time with $O(n)$ processors using $O(n)$ space on a EREW PRAM. We present algorithms $Maxima_1$ and $Maxima_2$ which optimally certify the 3-dimensional maxima problem.

Given a point p , let $xcor(p)$, $ycor(p)$, and $zcor(p)$ be the point's x coordinate, y coordinate, and z coordinate respectively. Given a horizontal line segment h , let $xleft(h)$ and $xright(h)$ denote the leftmost and rightmost x coordinates of the segment respectively. Given a vertical line segment v , let $ytop(v)$ and $ybottom(v)$ denote the segment's highest and lowest y coordinates respectively. Let $YMAX$ be the maximum y coordinate of all of the points in P .

The first phase works as follows. First, it finds the maximal points in S . Using the maximal points it decomposes the XY plane into regions, such that each region is associated with a point in P . This decomposition has the property that if a new point p were added to P , p would be maximal if and only if p had a larger z coordinate than the point whose region p falls into. See Figure 5.3 for an example. This decomposition is output as part of the certification trail. Also, for every maximal point, we perturb the point by adding a small positive number to both its X and Y coordinate, and output the region in which the perturbed point lies. Finally, for every point p which is not maximal, a point which dominates p is output along with p to prove that it is not maximal.

Algorithm $Maximal_1$: we are given a set of points P .

1. Find the set M of maximal points in P , and the set of D of the dominated points of P . For every $d \in D$, let $Dom(d)$ be a point which dominates d .
2. For every point $m \in M$, find the point $MaxX(m)$ which has the largest x coordinate of all points which dominate m in the YZ plane.
3. For every point $m \in M$, find the point $MaxY(m)$ which has the largest y coordinate of all points which dominate m in the XZ plane.
4. For every point $m \in M$, find the point $MaxZ(m)$ which has the largest z coordinate of all points which dominate m in the XY plane.
5. Construct a set H of horizontal line segments, a set V of vertical line segments, and a set Q of vertical rays as follows. For each $m \in M$ perform the following steps:
 - (a) If $MaxX(m) = nil$ then add the segment $\overline{(\theta, ycor(m)), (xcor(m), ycor(m))}$ to H , else add the segment $\overline{(xcor(MaxX(m)), ycor(m)), (xcor(m), ycor(m))}$ to H .
 - (b) If $MaxY(m) = nil$ then add the segment $\overline{(xcor(m), \theta), (xcor(m), ycor(m))}$ to V , else add the segment $\overline{(xcor(m), ycor(MaxY(m))), (xcor(m), ycor(m))}$ to V .
 - (c) If $MaxZ(m) = nil$ then add the ray $\overrightarrow{(xcor(m), ycor(m)), (xcor(m), YMAX + \epsilon)}$ to Q , else add the ray $\overrightarrow{(xcor(m), ycor(m)), (xcor(m), ycor(MaxZ(m)))}$ to Q .
6. Compute $IsoIntersect(H, V \cup Q) = (C, C_{\square})$.
7. Output $(M, (D, Dom, MaxX, MaxY, MaxZ, C_{\square}))$.

Refer again to Figure 5.3a. We see that the horizontal lines in the figure are precisely the segments in H , and the vertical lines in the figure are precisely the segments in V . In Figure 5.3b, the dashed vertical rays are precisely the rays in Q . For every point $m \in M$ there is a segment $m_h \in H$, a segment $m_v \in V$, and a ray $m_q \in Q$. We say that m_h and m_v are the segments associated with m , and that m_q is the ray associated with m . Recall that we consider all segments to be open, thus segments can only intersect at interior points.

Algorithm $Maximal_2$: we are given $(I, \tilde{M}, (\tilde{D}, \tilde{Dom}, \tilde{MaxX}, \tilde{MaxY}, \tilde{MaxZ}, \tilde{C}_{\square}))$ as input.

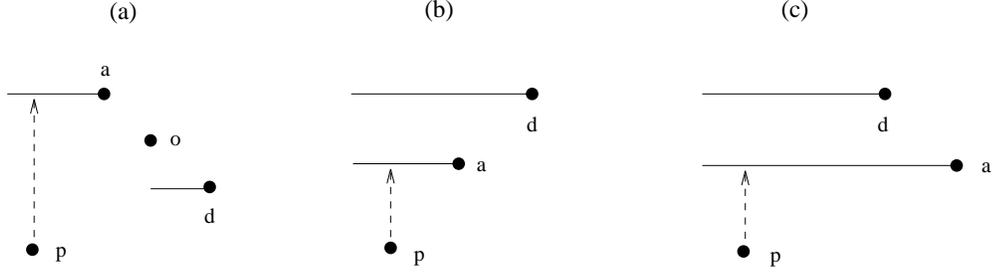


Figure 5.4: Illustrations for lemma 4.2.5.

1. Check that $\tilde{D} \cup \tilde{M} = I$.
2. For each $d \in \tilde{D}$, check that $\tilde{D}om(d)$ dominates d .
3. For each $m \in \tilde{M}$, check that $\tilde{M}axX(m)$ is a point which dominates m in the YZ plane, but has a smaller x coordinate than m .
4. For each $m \in \tilde{M}$, check that $\tilde{M}axY(m)$ is a point which dominates m in the XZ plane, but has a smaller y coordinate than m .
5. For each $m \in \tilde{M}$, check that $\tilde{M}axZ(m)$ is a point which dominates m in the XY plane, but has a smaller z coordinate than m .
6. For each $m \in \tilde{M}$, check that $\tilde{M}axX(\tilde{M}axZ(m)) < xcor(m) < xcor(\tilde{M}axX(m))$.
7. Construct \tilde{H} , \tilde{V} and \tilde{Q} as above, using the labels $\tilde{M}axX$, $\tilde{M}axY$, $\tilde{M}axZ$.
8. Check that $IsoIntersect_2((\tilde{H}, \tilde{V} \cup \tilde{Q}), nointer, \tilde{C}_{\square}) = ok$.
9. If all the above checks succeed output *ok*. Else output *error*.

Lemma 5.2.7: Consider a point $p = (x, y, z)$ in 3-space. Let $above(p)$ be the point associated with the first segment in H which intersects the ray $\overrightarrow{(x, y), (x, MAXY + \epsilon)}$. If no segment intersects the ray, then let $above(p) = (\infty, \infty, -\infty)$ by default. Then p is dominated by a point in P if and only if z is smaller than the z coordinate of the point $above(p)$.

Proof: Let p be a point. The first possibility is that $above(p) \neq (\infty, \infty, -\infty)$. Let $a_h \in H$ be the first segment in H which intersects the ray

$\overrightarrow{(xcor(p), ycor(p)), (xcor(p), MAXY + \epsilon)}$ in H . Let a be the point associated with a_h .

So assume that p is not maximal. By way of contradiction, let $zcor(p) > zcor(a)$ hold. Let d be the point with smallest x coordinate which dominates p . The first case is that $ycor(d) < ycor(a)$. See Figure 5.4a. But this implies that there is a point o such that $xcor(p) < xcor(o) < xcor(d)$, $ycor(d) < ycor(o)$, and $zcor(d) < zcor(o)$. But then o dominates p , which contradicts the hypothesis that d was the point with least x coordinate which dominated p . The second case is that $ycor(d) > ycor(a)$. The first sub-case is that $xcor(a) < xcor(d)$. See Figure 5.4b. But then $zcor(a) > zcor(d)$ or else a is not a maximal point. But this contradicts the hypothesis that $zcor(a) < zcor(p)$. The second sub-case is that $xcor(a) > xcor(d)$. See Figure 5.4c. Since $zcor(p) > zcor(a)$, $zcor(d) > zcor(p)$, $zcor(d) > zcor(a)$ must hold. But this is a contradiction because of the way a_h was constructed, as $zcor(d) > zcor(a)$ and $ycor(d) > ycor(a)$, and hence $xleft(a) \geq xcor(d)$ must hold.

Next assume that p is maximal. Since $xcor(a) > xcor(p)$ and $ycor(a) > ycor(p)$ both hold, it is clear that $zcor(a) < zcor(p)$ must also be true.

The other possibility is that $above(p) = (\infty, \infty, -\infty)$. By way of contradiction, assume that p is not maximal. Let d be the point with minimum x coordinate which dominates p . Consider the segment d_h . We claim that $xleft(d_h) < xcor(p)$. If this were not true then there would be a point d' which dominated p such that $xcor(d') < xcor(d)$ would hold. But $xleft(d_h) > xcor(p)$ must also be true, else the ray $\overrightarrow{(xcor(p), ycor(p)), (xcor(p), MAXY + \epsilon)}$ would intersect a segment in H . ■

Lemma 5.2.8: *Assume $Maxima_2$ returns ok. Then $\tilde{H} = H$ must hold.*

Proof: By way of contradiction, let $m \in \tilde{M}$ be a point which generates the line segment $m_h \in \tilde{H}$, such that at least one of the coordinates for m_h is incorrect.

Let us assume that m is the point with largest x coordinate which violates $\tilde{H} = H$. Let $m_h = \overline{(x_1, y_1), (x_2, y_2)}$. Let $c = MaxX(m)$. First assume that $c \neq nil$. Thus m_h should be of the form $\overline{(xcor(c), ycor(m)), (xcor(m), ycor(m))}$. By the way \tilde{H} is constructed, it is clear that y_1 , x_2 , and y_2 must all be correct. So first assume that $x_1 > xcor(c)$. So there exists a point w such that $xcor(w) = x_1$, $ycor(w) > ycor(m)$ and $zcor(w) > zcor(m)$. But then we see that $xcor(w) \leq xcor(MaxX(m))$ must hold, which is a contradiction. See Figure 5.5a.

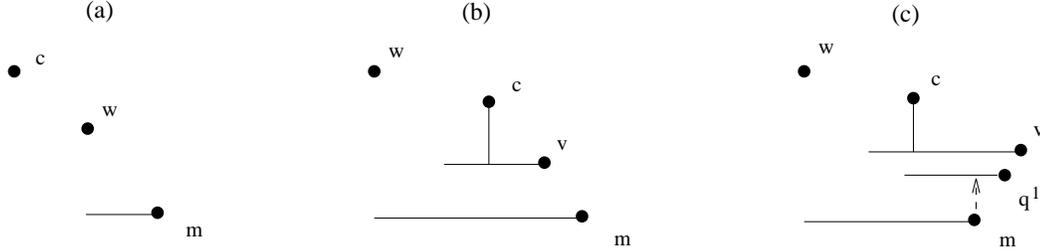


Figure 5.5: Illustrations for lemma 4.2.6.

So now assume that $x_l < xcor(c)$. Let $c_v \in V$ be the vertical segment associated with c . Clearly, $ybottom(c_v) > ycor(m)$, else h and c_v will intersect. Thus there is a point $v \neq m$, such that $zcor(v) > zcor(c)$, $ycor(c) > ycor(v) > ycor(m)$ and $xcor(v) > xcor(c)$. If $xcor(v) < xcor(m)$, then $xcor(v) \leq xcor(MaxX(m))$ must hold, which is again a contradiction. See Figure 5.5b.

Thus $xcor(v) > xcor(m)$ must hold. See Figure 5.5c for this case. Associated with m is a query ray m_q . Thus, there exists a point q^1 such that $ytop(m_q) = ycor(q^1)$ and $zcor(q^1) < zcor(m)$. By check 6, we see that $xcor(MaxX(q^1)) < xcor(m) < xcor(q^1)$. Since $zcor(m) < zcor(c) < zcor(v)$ is true, $zcor(m) < zcor(v)$ must also be true. If $q^1 = v$ we are done, because then we see that check 5 would have failed. So $q^1 \neq v$. Now we claim that $xcor(q^1) < xcor(v)$. This is because v dominates q^1 in the YZ plane, and hence $MaxX(q^1) \geq xcor(v)$ would hold, which would violate our assumption that m was the rightmost point whose associated horizontal segment was incorrect. So consider the ray q_q^1 . As before, there must be a point q^2 such that $ycor(q^2) = ytop(q_q^1)$. All of relations between m and q^1 must also hold between q^1 and q^2 . Furthermore, both $q^2 \neq v$ and $xcor(q^2) < xcor(v)$ must also hold. We keep iterating this way, producing the sequence of points q^1, q^2, q^3, \dots which is a contradiction, since $zcor(q^i) < zcor(q^{i+1})$ for all i and we have only a finite number of points.

The case that $MaxX(m) = nil$ is similar and is omitted. ■

Lemma 5.2.9: *Algorithm $Maximal_1$ and $Maximal_2$ certify the 3-dimensional dominance problem, and have the following properties: properties: (i) $Maximal_1$ runs in*

$O(\log n)$ time with $O(n)$ processors on a CREW PRAM, (ii) $Maxima_2$ runs in $O(1)$ time with $O(n)$ processors on a polling CREW PRAM.

Proof: The correctness of $Maxima_1$ and $Maxima_2$ follows from lemmas 5.2.7 and 5.2.8. The complexity follows from lemma 5.2.6 and the results of [5]. ■

5.3 Other Common Parallel Problems

Throughout this section, we construct and perform operations on binary trees. For consistency, if a node has only a single child, we consider the child to be the left child of the parent node. Thus, the inorder traversal of the trees we consider is well defined.

First we define the lowest common ancestor (LCA) problem [71]. Given a rooted tree, we wish to answer queries on the tree of the following form. Given two nodes u and v in the tree, find the node w with greatest depth that is an ancestor of both u and v .

Next we define the range maxima queries problem. We are given a list (a_1, a_2, \dots, a_s) of numbers and also query intervals $([l_1, r_1], [l_2, r_2], \dots, [l_t, r_t])$. For each query interval $[l_i, r_i]$ we wish to find the largest number in the set $\{a_{l_i}, a_{l_i+1}, \dots, a_{r_i}\}$. [38] shows how to solve this problem sequentially using a *cartesian tree* [83]. The cartesian tree on (a_1, a_2, \dots, a_s) is the heap ordered tree whose nodes are from $\{a_1, a_2, \dots, a_s\}$, such that an inorder traversal of the tree yields (a_1, a_2, \dots, a_s) . A tree is heap ordered if when a node a_u is the parent of node a_v , then $a_u \geq a_v$ must hold. Given a cartesian tree, the maximum number in the set $\{a_i, a_{i+1}, \dots, a_j\}$ is easy to find: if a_u is the LCA of a_i and a_j in the cartesian tree, then the maximum number is a_u .

Before describing how we certify the range maxima queries problem we present algorithms $LcaQ_1$ and $LcaQ_2$ for certifying LCA queries, and algorithms $Cart_1$ and $Cart_2$ for certifying the problem of constructing cartesian trees. Since there exist work-optimal parallel algorithms for these problems, the general simulation lemma given at the start of this chapter can be used to certify these problems. However, the techniques that we present here are more direct, and would most likely involve better constants in the second phases for these problems.

We use the following definitions and notation throughout the rest of this section. Assume we are given a binary tree Γ with each node labeled with an integer.

For each node $v \in \Gamma$, let $l(v)$ be the label of v . Also, For each node $v \in \Gamma$ we define $\min(v)$ to be the smallest label in the subtree rooted at v , and $\max(v)$ to be the largest label in subtree rooted at v .

Lemma 5.3.1: *Let Γ be a binary tree. Also, assume that for each node $v \in \Gamma$, we are given the values $\tilde{\min}(v)$ and $\tilde{\max}(v)$. Assume the following checks succeed:*

1. *That for the root r of Γ , $\tilde{\min}(r) = 1$ and $\tilde{\max}(r) = n$.*
2. *For every node v , $\tilde{\min}(v) \leq \tilde{\max}(v)$.*
3. *For every node v which is a leaf, $\tilde{\min}(v) = \tilde{\max}(v) = l(v)$.*
4. *For every node v with only one child w , $\tilde{\min}(v) = \tilde{\min}(w)$, $\tilde{\max}(v) = \tilde{\max}(w) + 1$, and $l(v) = \tilde{\max}(v)$.*
5. *For every v with left child u and right child w , $\max(u) + 2 = \tilde{\min}(w)$, $\tilde{\min}(v) = \tilde{\min}(u)$, $\tilde{\max}(v) = \tilde{\max}(w)$, and $l(v) = \tilde{\max}(u) + 1$.*

Then, the labels in an inorder traversal of Γ yield $(1, 2, \dots, n)$. Moreover, for all $v \in \Gamma$ we have $\tilde{\min}(v) = \min(v)$ and $\tilde{\max}(v) = \max(v)$.

Proof: This lemma can be shown by induction on the heights of the nodes in Γ . ■

We call the five checks listed in the above lemma the *InOrderChecks*.

Lemma 5.3.2: *Let Γ be a binary tree such that the labels in an inorder traversal of the nodes yield $(1, 2, \dots, n)$. Then a node u is an ancestor of a node v if and only if $\min(u) \leq l(v) \leq \max(u)$.*

Proof: This lemma can also be shown using induction. ■

Algorithm $LcaQ_1$: We are given $(\Gamma, ((u_1, v_1), (u_2, v_2), \dots, (u_t, v_t)))$ as input. Γ is labeled in an inorder fashion.

1. Compute (a_1, a_2, \dots, a_t) as the answers to the LCA queries on Γ .
2. For each node $v \in \Gamma$, compute $\min(v)$ and $\max(v)$.
3. Output $((a_1, a_2, \dots, a_t), ((\min(1), \min(2), \dots, \min(n)), (\max(1), \max(2), \dots, \max(n))))$.

Algorithm $LcaQ_2$: We are given $(\Gamma, ((u_1, v_1), (u_2, v_2), \dots, (u_t, v_t)))$ as input, $(\tilde{a}_1, \tilde{a}_2, \dots, \tilde{a}_t)$ as the answers, and $((\tilde{min}(1), \tilde{min}(2), \dots, \tilde{min}(n)), (\tilde{max}(1), \tilde{max}(2), \dots, \tilde{max}(n)))$ as the certification trail.

1. Perform the *InOrderChecks* on the nodes in Γ and the \tilde{min} and \tilde{max} labels.
2. For each answer \tilde{a}_i to query (u_i, v_i) , check that $\tilde{min}(a_i) \leq l(u_i) \leq \tilde{max}(a_i)$, and that $\tilde{min}(a_i) \leq l(v_i) \leq \tilde{max}(a_i)$.
3. For each child a'_i of \tilde{a}_i , check that either $\tilde{min}(a'_i) \leq l(u_i) \leq \tilde{max}(a'_i)$ or $\tilde{min}(a'_i) \leq l(v_i) \leq \tilde{max}(a'_i)$ does not hold.
4. If all of the checks succeed output *ok*, else output *error*.

Lemma 5.3.3: *Algorithms $LcaQ_1$ and $LcaQ_2$ certify the lowest common ancestors problem and have the following properties: (i) $LcaQ_1$ runs in $O(\log |\Gamma|)$ time performing linear work on an CREW PRAM, (ii) $LcaQ_2$ runs in $O(1)$ time performing linear work on a polling CREW PRAM.*

Proof: The correctness of $LcaQ_1$ and $LcaQ_2$ follows from lemmas 5.3.1 and 5.3.2.

The implementation of $LcaQ_1$ is performed using the results of [71] and [55]. The implementation of $LcaQ_2$ is straightforward. ■

Next we describe algorithms $Cart_1$ and $Cart_2$ for certifying the problem of constructing the cartesian tree.

Algorithm $Cart_1$: We are given (a_1, a_2, \dots, a_s) as input.

1. Construct the cartesian tree Υ . Label each node a_i with i .
2. For each node $v \in \Upsilon$, compute $min(v)$ and $max(v)$.
3. Output $(\Upsilon, (min(1), min(2), \dots, min(s)), (max(1), max(2), \dots, max(s)))$.

Algorithm $Cart_2$: We are given (a_1, a_2, \dots, a_s) as input, a labeled tree $\tilde{\Upsilon}$ as the answer, and $((\tilde{min}(1), \tilde{min}(2), \dots, \tilde{min}(s)), (\tilde{max}(1), \tilde{max}(2), \dots, \tilde{max}(s)))$ as the certification trail.

1. Check that $\tilde{\Upsilon}$ is a binary tree.
2. Check that for every node v , $a_{l(v)} = v$.

3. For each child w of a node v , check that $v \geq w$ holds.
4. Perform the *InOrderChecks* on $\tilde{\Gamma}$ with the \tilde{min} and \tilde{max} values.
5. If the checks succeed output *ok*, else output *error*.

Lemma 5.3.4: *Cart₁ and Cart₂ certify the problem of constructing a cartesian tree and have the following properties: (i) Cart₁ runs in $O(\log s)$ time performing linear work on a CREW PRAM, (ii) Cart₂ runs in $O(1)$ time performing linear work on a polling CREW PRAM.*

Proof: First we prove the correctness of *Cart₁* and *Cart₂*. It is easy to show that if *Cart₂* is given the output and the certification trail from *Cart₁*, then *Cart₂* outputs *ok*. Next assume that *Cart₂* outputs *ok*. Since the *InOrderChecks* succeed, we know that the labels in an inorder traversal of $\tilde{\Gamma}$ give the list $(1, 2, \dots, s)$. Hence, an inorder traversal of the tree yields the list (a_1, a_2, \dots, a_s) . Since $\tilde{\Gamma}$ is also heap ordered, $\tilde{\Gamma}$ must be the cartesian tree of (a_1, a_2, \dots, a_s) .

The implementation of *Cart₁* is performed using the results of [12]. The implementation of *Cart₂* is straightforward. ■

We are now ready to show how to certify the problem of answering minimum range queries over a list.

Algorithm Range₁: We are given list L and intervals I as input.

1. Compute $Cart_1(L) = (\Upsilon, \Upsilon_{\square})$.
2. Compute $Lca_1(\Upsilon, I) = (A, A_{\square})$.
3. Output $(A, (\Upsilon, \Upsilon_{\square}, A_{\square}))$.

Algorithm Range₂: We are given $((L, I), \tilde{A}, (\tilde{\Upsilon}, \tilde{\Upsilon}_{\square}, \tilde{A}_{\square}))$ as input.

1. Check that $Cart_2(L, \tilde{\Upsilon}, \tilde{\Upsilon}_{\square}) = ok$.
2. Check that $Lca_2(\tilde{\Upsilon}, I, \tilde{A}_{\square}) = ok$.
3. If both checks succeed output *ok*, else output *error*.

Lemma 5.3.5: *Range₁ and Range₂ certify the range maxima queries problem and have the following properties: (i) Range₁ runs in $O(\log s)$ time performing linear work on a CREW PRAM, (ii) Range₂ runs in $O(1)$ time performing linear work on a polling CREW PRAM.*

Proof: This lemma follows immediately from the correctness of $LcaQ_1$, $LcaQ_2$, $Cart_1$, $Cart_2$, and the previously mentioned properties of cartesian trees. ■

Chapter 6

Parallel Checking and Certifying of Minimum Spanning Trees

6.1 Introduction

In this chapter we consider the problem of checking that a tree T is a Minimum Spanning Tree of a given graph G . We present algorithms for both a linear array of processors and the PRAM.

Before discussing our results, we discuss the most significant previous results for the problem of checking the correctness of an MST. Throughout this chapter, let n be the number of nodes in the graph G , and m be the number of edges in G . In addition, let $N = n + m$. In [9], Alon and Schieber present an $O(n\lambda(n) + m)$ time sequential algorithm for checking MSTs, where $O(\lambda(n))$ is any function from a class of very slow growing functions (of which $O(\log^*(n))$, otherwise known as the iterated *log* function, is a member). They show how their algorithm can be implemented to run in $O(\log N)$ time on a PRAM, while performing $O(n\lambda(n) + m)$ work. Dixon, Rauch and Tarjan [31] show how MSTs can be checked in $O(N)$ sequentially. Recently, by using the Alon and Schieber result as a subroutine, they show how to implement their algorithm in $O(\log N)$ time with $O(N)$ work on a PRAM.

We present results for both a linear array of processors and the PRAM. For a graph G and tree T , we show how a linear array of n processors can check if T is an MST of G in $O(n)$ time. Although it is possible to implement the sequential algorithm of [31] in the same bounds on a linear array, our algorithm is conceptually much simpler than their algorithm, and yields good constants. The authors themselves view their algorithm more as a theoretical result than as a program that would yield good results in practice [32].

We present a certifier that runs in constant time with $O(N)$ processors on a PRAM. In addition to running in constant time, the checker has the further advantage of being quite simple.

6.2 Definition of Prim Sequence

First we will review how Prim's algorithm, a sequential algorithm for generating the MST of a graph works. Initially, a vertex r in V is chosen arbitrarily and will be the root vertex of the MST which is output. The algorithm also uses a set S called the *Prim set* which initially contains only r . Then until $S = V$ we repeat the following

procedure: find an edge of minimum weight which connects a vertex in S to a vertex not in S . Output this edge, and then add the second vertex to S . Notice that if the edge weights are unique, then there is only one order that Prim may choose edges starting from a particular vertex. Also notice that if the edge weights are not unique then there may be more than one order in which Prim's algorithm may choose the edges.

We note that for a graph of $G = (V, E)$, Prim's algorithm can be implemented using a priority queue to run in $O(|E| \log |V|)$ time. By using a specialized type of priority queue known as a Fibonacci Heap, [36] show that this can be reduced to $O(|V| \log |V| + |E|)$ time.

To verify whether T is the MST of G we attempt to show whether Prim's algorithm could have output T . We will need some definitions.

Definition 1: Let $G = (V, E, W)$ be a weighted graph, and $v \in V$. A Prim vertex sequence of G starting from v is an ordered sequence of the vertices of V which corresponds to the order in which they are added to the Prim set S during some execution of Prim's algorithm when v is chosen as the starting vertex. This sequence does not include the vertex v itself. A Prim edge sequence is an ordered sequence of a subset of E which corresponds to the order in which they are selected during some execution of Prim's algorithm. When there is no ambiguity we will refer to both vertex and edge sequences as Prim sequences. When referring to a Prim sequence we will use a subscript to indicate the starting vertex used by Prim's algorithm.

Definition 2: Let $G = (V, E)$ be a graph, and $v \in V$. We define a tree vertex sequence of G starting from v as an ordered sequence of the vertices of V corresponding to a run of Prim's algorithm on G starting from v when all of the weights of the edges are assumed to be the same. A tree edge sequence is an ordered sequence of a subset of E which corresponds to the order in which they are selected during some execution of Prim's algorithm when all edges are assumed to have the same weight.

Lemma 6.2.1: Let $G = (V, E, W)$ be a weighted graph, $T \subset E$ be a MST of G , and $v_0 \in V$. Then if O_{v_0} is a Prim vertex sequence of (V, T, W) , then O_{v_0} is a Prim vertex sequence of (V, E, W) as well. Similarly, if O_{v_0} is a Prim edge sequence of (V, T, W) , then O_{v_0} is a Prim edge sequence of (V, E, W) as well.

Proof: Let $O_{v_0} = (v_1, v_2, \dots, v_{n-1})$ be a Prim vertex sequence of (V, T, W) which is not a Prim vertex sequence of (V, E, W) . Let $O'_{v_0} = (v'_1, v'_2, \dots, v'_{n-1})$ be a Prim vertex sequence of (V, E, W) such that no other Prim sequence of (V, E, W) starting from v_0 has a larger prefix in common with O_{v_0} . Let (e_1, \dots, e_{n-1}) and (e'_1, \dots, e'_{n-1}) be the Prim edge sequences which correspond to O_{v_0} and O'_{v_0} . Let j be the smallest index such that $v_j \neq v'_j$. We claim that $W(e'_j) < W(e_j)$. If the weights were equal then there is another Prim sequence of (V, E, W) with a larger prefix in common with O_{v_0} . If the weight of e'_j was greater than the weight of e_j then O'_{v_0} is not a Prim sequence since the algorithm would have chosen e_j instead of e'_j while generating O'_{v_0} . But now consider what would happen if e'_j were added to T . A cycle would be formed, and an edge e on this cycle besides e'_j must connect a vertex in $\{v_0, v_1, \dots, v_{j-1}\}$ to a vertex in $\{v_j, \dots, v_{n-1}\}$. But $W(e) \geq W(e_j)$ else Prim's algorithm would have chosen e while running on (V, T, W) . If we remove e and add e'_j we would have a spanning tree of total weight smaller than T . This is a contradiction since T was assumed to be a MST, and the lemma is proved. ■

An immediate corollary of this lemma is that any MST of a graph can be generated by some execution of Prim's algorithm regardless of which vertex is chosen to be the start vertex.

Given $G = (V, E, W)$, tree $T \subset E$ and some vertex $v \in V$ let O_v be a Prim edge sequence of (V, T, W) and let O'_v be the corresponding Prim vertex sequence. By lemma 3.1, if T is a MST then O_v is a Prim sequence of G . By definition, if O_v is a Prim sequence of G then T is a MST of G . The following two lemmas prove to be useful.

Lemma 6.2.2: *Let $O_{v_0} = (e_1, \dots, e_{n-1})$ and $O'_{v_0} = (v_1, \dots, v_{n-1})$. If for every edge $e_i = (v_j, v_k) \in O_{v_0}$ (with $j < k$) we have $k = i$, then the edges in O_{v_0} are a tree, and O_{v_0} and O'_{v_0} are corresponding tree sequences.*

Proof: We can prove by induction that for all $1 \leq i \leq n-1$, e_1, \dots, e_i is a tree, and that (e_1, \dots, e_i) and (v_1, \dots, v_i) are corresponding tree sequences. ■

Lemma 6.2.3: *Let $O_{v_0} = (e_1, e_2, \dots, e_{n-1})$ and $O'_{v_0} = (v_1, v_2, \dots, v_{n-1})$ be corresponding tree sequences. If for all edges $(v_i, v_j) \in E$ (with $i < j$) and for all edges $e_k \in O_{v_0}$ with $i < k \leq j$ we have $W((v_i, v_j)) \geq W(e_k)$ then O_{v_0} and O'_{v_0} are corresponding Prim sequences of G .*

Proof: Again, we can prove by induction that for all $1 \leq i \leq n-1$ (v_1, \dots, v_i) and (e_1, \dots, e_i) are corresponding Prim sequences of the subgraph of G induced by $\{v_1, \dots, v_i\}$. ■

Lemma 6.2.4: Let $O_{v_0} = (e_1, e_2, \dots, e_{n-1})$ and $O'_{v_0} = (v_1, v_2, \dots, v_{n-1})$ be corresponding tree sequences. If O_{v_0} and O'_{v_0} are corresponding Prim sequences of G , then for all edges $(v_i, v_j) \in E$ (with $i < j$) and for all edges $e_k \in O_{v_0}$ with $i < k \leq j$ we have $W((v_i, v_j)) \geq W(e_k)$.

Proof: This lemma is the converse of the previous one, and is true by the definition of a Prim sequence. ■

Lemma 6.2.5: Let $G = (V, E)$ be a graph, and let $T \subset E$ be a spanning tree of G . Suppose $E = E_1 \cup E_2 \cup \dots \cup E_k$. Then T is an MST of G if and only if for all $1 \leq i \leq k$, T is an MST of the graph $(V, E_i \cup T)$.

We now define the *range maxima queries* problem. We are given an array $A = (a_1, \dots, a_s)$ of s numbers and also t query intervals $I = ([l_1, r_1], [l_2, r_2], \dots, [l_t, r_t])$. For each query interval $[l_i, r_i]$ we wish to find the largest number in the set $\{a_{l_i}, a_{l_i+1}, \dots, a_{r_i}\}$. We use $MRQ([i, j], A)$ to denote the maximum number in the array A which lies in the interval defined by $[i, j]$. We also define the lower envelope problem. We are given t weighted intervals $I = ([l_1, r_1], [l_2, r_2], \dots, [l_t, r_t])$ over the integers $(1, 2, \dots, n)$. For each $1 \leq j \leq n$, we wish to find the minimum weight of all of the intervals which contain j , which we denote $LEQ(j, I)$.

These two definitions suggest the following two algorithms for checking MSTs.

Algorithm RangeCheckMST:

1. Pick an arbitrary vertex v_0 from V .
2. Construct the Prim sequences $O_{v_0} = (v_1, v_2, \dots, v_{n-1})$ and $O'_{v_0} = (e_1, e_2, \dots, e_{n-1})$.
3. For every edge $(v_i, v_j) \in E$ (with $i < j$), check that $RMQ([i+1, j], (W(e_1), W(e_2), \dots, W(e_{n-1}))) \leq W(v_i, v_j)$.
4. If all these check succeed, output *ok*, else output *error*.

Algorithm EnvelopeCheckMST:

1. Pick an arbitrary vertex v_0 from V .
2. Construct the Prim sequences $O_{v_0} = (v_1, v_2, \dots, v_{n-1})$ and $O'_{v_0} = (e_1, e_2, \dots, e_{n-1})$.
3. Construct the set $I = \{[i+1, j] \text{ (with weight } W(v_i, v_j)) \mid (v_i, v_j) \in E\}$.
4. For each $1 \leq j \leq n-1$, check that $LEQ(j, I) \geq W(e_j)$.
5. If all these check succeed, output *ok*, else output *error*.

6.3 Checking on a Linear Array of Processors

We now discuss how given a graph $G = (V, E)$ and $T \subset E$, how to determine if T is an MST of G . Our algorithm runs on a linear array of $|V|$ processors in $|V|$ time. The algorithm is based on *EnvelopeCheckMST* presented in the last section.

We now discuss the input representation used by our algorithm. Let $n = |V|$. Let $V = \{0, 1, \dots, n-1\}$ be the vertices. Let P_0, P_2, \dots, P_{n-1} be the processors. An edge (i, j) can be stored in processor P_i , processor P_j , or possibly both processors. We assume that all of the edges of T are stored in P_0 . We construct a Prim sequence of T , which is broadcasted to all of the processors. For each processor P_i , we construct the lower envelope of the edges stored in the processor. Since all of the edges stored at P_i are adjacent to vertex i , the lower envelope is unusually simple to compute.

First we present algorithm *ComputeEnvelope*.

Algorithm *ComputeEnvelope*: We are given a set of t weighted intervals $I = ([l_1, r_1], [l_2, r_2], \dots, [l_t, r_t])$ over the point set $(0, 1, \dots, n-1)$. We are also given a value X , and we assume that for each interval in I , that $l_i = X$ or $r_i = X$. We wish to compute $LEQ(j, I)$ for $1 \leq j \leq n$. We store all of the answers in the array *leq_ans*.

1. For $0 \leq j \leq n-1$, set $leq_ans[j] = 0$.
2. For every interval of the form $[X, r_i] \in I$, set $leq_ans[r_i] = W([X, r_i])$.
3. For every interval of the form $[l_i, X] \in I$, set $leq_ans[l_i] = W([l_i, X])$.
4. For every $j \leq X$, set $leq_ans[j] = \text{MIN}_{k=0}^j leq_ans[k]$.
5. For every $X \leq j$, set $leq_ans[j] = \text{MIN}_{k=j}^{n-1} leq_ans[k]$.

Lemma 6.3.1: *Procedure *ComputeEnvelope* is correct, and runs in $O(t+n)$ time.*

Procedure *CheckMST*:

- 1 P_0 checks that the edges in T are a spanning tree of V .
- 2 P_0 finds the Prim sequence of T starting from an arbitrary vertex v_0 .
Let $O_{v_0} = (v_1, v_2, \dots, v_{n-1})$ and $O'_{v_0} = (e_1, e_2, \dots, e_{n-1})$ be the corresponding Prim vertex and edge sequences constructed.
- 3 Broadcast O_{v_0} and O'_{v_0} to all of the nodes in the linear array.

For the remaining steps we restrict our attention to the computation in the processor P_i .

- 4 Let E_i be the edges stored in P_i .
- 5 Let $E_i^1 = \{(v_{j_l} + 1, i) \mid (v_{j_l}, i) \in E_i \text{ and } v_{j_l} < i\}$.
- 6 Let $E_i^2 = \{(i + 1, v_{j_l}) \mid (v_{j_l}, i) \in E_i \text{ and } v_{j_l} > i\}$.
- 7 For each $1 \leq l \leq n$, check that $LEQ(E^1, l) \geq W(e_l)$ and $LEQ(E^2, l) \geq W(e_l)$.
- 8 If all these checks succeed output *ok*, else output *error*.

Lemma 6.3.2: *Algorithm CheckMST is correct, and runs in $O(n)$ time on a $O(n)$ linear array of processors.*

Proof: The correctness of *CheckMST* follows directly from lemmas 1.1 and 2.2. The only difficulty is the construction of the Prim sequences. For this we use the systolic priority queue algorithm. For an n node systolic array, it is possible to maintain a priority queue with $O(n)$ elements so that each priority queue operation requires $O(1)$ time. Thus we run Prim's algorithm on T , using the systolic priority queue implementation, achieving an $O(n)$ running time. The rest of the algorithm clearly takes $O(n)$ time. ■

6.4 PRAM Implementation

6.4.1 Finding the Prim Sequence of a Tree in Parallel

Finding the Prim sequence of a tree is easier to explain and perform when the tree is binary. Hence, we will show how to convert an arbitrary tree T into a binary tree T_b such that given a Prim sequence of T_b we can determine a Prim sequence of T . Let δ be the smallest weight given to an edge divided by 2. For every node $v \in T$ with children c_1, \dots, c_k there will be the nodes $v, v_2, \dots, v_k, c_1, \dots, c_k$ in T_b . There will be an edge of weight δ between v and v_2 , and also for all $2 \leq i \leq k - 1$ there will be an

edge between v_i and v_{i+1} of weight δ . The edge (v, c_1) in T_s is in T_b and for all i there will be an edge (v_i, c_i) of the same weight as (v, c_i) . It is not hard to show that the transformation can be performed with $O(n)$ work in $O(\log n)$ time.

Lemma 6.4.1: *Let O_v be a Prim vertex sequence for T_b . Let O'_v be obtained from O_v by removing all vertices not in T . Then O'_v is a Prim vertex sequence of T .*

This lemma is easy to show and it is also clear that the Prim vertex sequence for T can be obtained from the Prim vertex sequence with $O(n)$ work in $O(\log n)$ time.

Note that we can also assume that the edge weights in T_b are unique. Any ties in the edge weights can be broken consistently using the memory locations where the edges are stored for a secondary comparison. By breaking ties among edge weights Prim's algorithm becomes deterministic, and thus every graph has exactly one Prim sequence (when starting from the same vertex).

We will now show how to transform the problem of finding the Prim sequence of T_b into the problem of evaluating a special type of operator tree. We refer to one of the operators in our tree as the generalized merge operator. It is defined below.

Definition 3: *Let L_1 and L_2 be two lists of numbers such that no element appears in both lists. We define the generalized merge, denoted as $L = L_1 \otimes L_2$ as the result of the following procedure: initially beginning with the complete lists L_1 and L_2 , we compare the first remaining element in L_1 to the first remaining element in L_2 . If number in L_1 is smaller than the number in L_2 we move the first remaining element in L_1 to the next vacant spot in L . Else we move the first remaining element in L_2 into the next vacant spot of L . This is repeated until either L_1 or L_2 are empty. At this point we move the remaining elements of the other list onto the end of L . Note, we sometimes use the term merge to abbreviate generalized merge.*

Notice that this is just the standard sequential merge procedure without the requirement that the inputs to the merge be sorted. When we apply the operator \otimes on lists of edges we have the understanding that it is the weight of the edges which is to be used for the comparisons.

Definition 4: *A concatenate-merge tree is an ordered binary tree with either the operator \otimes or $|$ at each internal node. Each node must have exactly two children.*

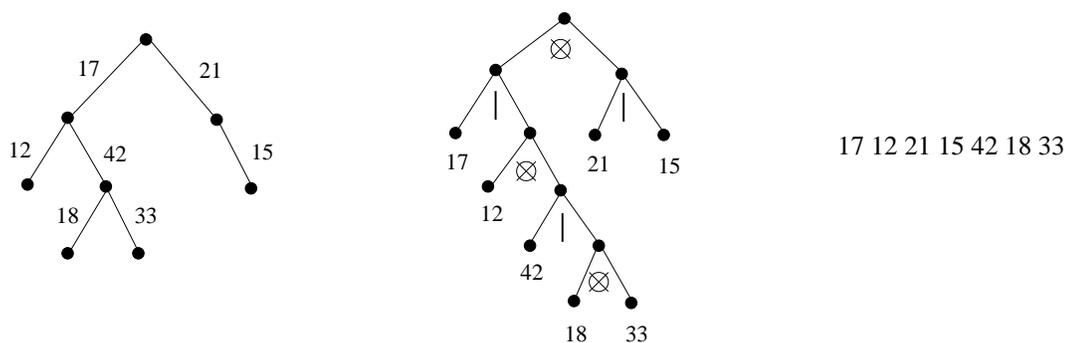


Figure 6.1: Construction of concatenate-merge tree from a weighted spanning tree.

Each leaf has a list containing one number, and all of the numbers at the leaves must be distinct. The operator \otimes is defined as before. The operator $|$ is called concatenate, and if L_1 and L_2 are two lists, $L_1 | L_2$ is defined to be the list with L_2 added onto the end of L_1 . The value of a node and of the tree are defined naturally.

Given a binary tree T_b which we wish to find a Prim sequence of, it is possible to construct a concatenate-merge tree which yields such a sequence.

We now give a recursive construction of T_{op} from T_b . Let r be the root node of T_b .

Assume r has only one child c . If c is a leaf node then T_{op} consists only of a root node with the value $W((r, c))$ stored at the node.

Else c is not a leaf node. Let T_c be the concatenate-merge tree for the subtree of T_b rooted at c . The node r_{op} will be the root of the concatenate-merge tree for T_b . The operator at r_{op} is $|$. The node c_l will be the left child of r_{op} , and we give c_l the value $W((r, c))$. Finally, the right child of r_{op} will be the root of T_c .

Now assume that r has two children c and c' . The node r_{op} will be the root node of concatenate-merge tree of T_b . The operator at r_{op} is \otimes . r_{op} has children c_l and c_r . If c is a leaf node then c_l is a leaf node and stores the value $W((r, c))$. Else if c is not a leaf node then let T_c be the concatenate merge tree for the subtree of T_b rooted at c and perform the following 3 steps: (i) give c_l the operator $|$, (ii) let c_l' be the left child of c_l and give c_l' the value $W((r, c))$, (iii) make T_c be the right child of c_l . Similarly, if c' is a leaf node then c_r is a leaf node and stores the value $W((r, c'))$. Else if c' is not a leaf node then let $T_{c'}$ be the concatenate merge tree

for the subtree of T_b rooted at c' and perform the following 3 steps: (i) give c_r the operator $|$, (ii) let $c_{r'}$ be the left child of c_r and give $c_{r'}$ the value $W((r, c'))$, (iii) make $T_{c'}$ be the right child of c_r . See Figure 6.1 for an example of this construction. The first part of the figure is a weighted spanning tree, the second part of the figure is the corresponding concatenate-merge tree, and the third part of the figure is the resulting Prim sequence.

Lemma 6.4.2: *The value of T_{op} as constructed above yields the Prim edge sequence of T_b .*

We give the proof of this lemma in the final version of the paper. Also, it is not hard to show that this construction can be performed in $O(\log n)$ time with $O(n)$ work.

6.4.2 Evaluating the Concatenate-Merge Tree

Before considering the evaluation of the concatenate-merge tree we consider the simpler problem of performing a generalized merge.

Algorithm Generalized Merge: We are given two lists $A = (a_1, \dots, a_s)$ and $A' = (a'_1, \dots, a'_s)$ of distinct numbers. We output $A \otimes A'$.

1. construct $I = (i_1, \dots, i_s)$ such that i_x is the index of the largest element in $\{a_1, \dots, a_x\}$. Similarly construct I' .
2. construct $L = ((a_{i_1}, i_1, a_1), (a_{i_2}, i_2, a_2), \dots, (a_{i_s}, i_s, a_s))$ and $L' = ((a'_{i'_1}, i'_1, a'_1), (a'_{i'_2}, i'_2, a'_2), \dots, (a'_{i'_s}, i'_s, a'_s))$
3. using lexicographic comparisons, compute $M = ((leader_1, index_1, value_1), \dots, (leader_{2*s}, index_{2*s}, value_{2*s}))$ of the merge of L and L' .
4. output $(value_1, \dots, value_{2*s})$ ■

Lemma: The above algorithm correctly computes $A \otimes A'$.

Proof: By way of contradiction, assume that the algorithm does not correctly compute $A \otimes A'$. Let $((l_1, i_1, v_1), \dots, (l_{2*s}, i_{2*s}, v_{2*s}))$ be the value of M after the merge in step 3 of the algorithm. Notice that both L and L' are monotonically increasing, thus the the standard merge on L and L' is well defined, and the relative

order of L and L' in M is preserved. Let $(ans_1, \dots, ans_{2*s})$ be the output of the sequential merge procedure applied to A and A' . By definition, $(ans_1, \dots, ans_{2*s}) = A \otimes A'$. Let j be the smallest index such that $v_j \neq ans_j$. Without loss of generality, assume that $(l_j, i_j, v_j) \in L$ (and hence originally comes from A). Thus, there is a k such that $(l_k, i_k, v_k) \in L'$ and $v_k = ans_j$. Clearly $k > j$, since M was correct up to position j . Thus $l_j < l_k$. But also $v_k < v_j$, since during step j of the sequential merge procedure both v_j and v_k were the first of the remaining elements in A and A' . Since $l_x \geq v_x$ for all x , we get $l_j \geq v_j$. Combining the facts $v_k < v_j$, $l_j < l_k$, and $l_j \geq v_j$, we get $l_k > v_k$. Therefore, there must exist an p such that $(l_p, i_p, v_p) \in L'$ with $p < k$ and $l_p = v_p = l_k$. But $p < j$ must also hold, since (l_k, i_k, v_k) is the first entry in M after position j from L' . But $p < j$ implies that $(l_p, i_p, v_p) < (l_j, i_j, v_j)$ since M was formed by merging sorted lists L and L' together. But this contradicts the earlier statements that $l_p = l_k > l_j$. ■

Given $A = (a_1, \dots, a_s)$, let $L = ((a_{i_1}, i_1, a_1), \dots, (a_{i_s}, i_s, a_s))$ be as in step 2 of the generalized merge algorithm. For all $1 \leq x \leq s$ such that $a_{i_x} = a_x$, we say a_x is a *leader* of A . Similarly we define the *leaders* of A' . Intuitively, the generalized merge algorithm works by breaking A and A' into blocks according to their leaders with the leader as the first element in each block. We say that the leader of a block *dominates* the other elements in the block. We then merge the blocks together by their leaders.

We now consider the problem of evaluating the concatenate-merge tree. We define the *depth* of a node v in a tree to be the number of vertices in the path from v to the root. Note that the root is of depth one.

Consider evaluating the nodes of T_{op} starting from the leaves and working towards the root. Initially, every element in the tree is a leader. As the computation progresses, more and more elements become dominated. Let u be a node in T_{op} with left child v and right child w . If N_1 is a leader in $val(v)$, then it must be a leader in $val(u)$. If N_1 is a leader in $val(w)$ then it may or may not be a leader in $val(u)$. If it is not then the operator at u must be $|$ and the largest number in the subtree of T_{op} rooted at v must be larger than N_1 . Using this characterization of when elements become dominated we now give a procedure for evaluating T_{op} .

Algorithm Evaluate T_{op} : We are given as input the operator tree T_{op} . We output the value of the tree.

1. For every internal node in T_{op} calculate the maximum of the numbers stored in the leaves of the subtree rooted at that internal node [55]. Assign this value to the label max at every internal node.
2. Let N_1 be stored at leaf l_1 . Let v_x, v_{x-1}, \dots, v_1 be the nodes on the path from l_1 to the root of T_{op} (with $v_x = l_1$ and $v_1 = root(T_{op})$). Let i be the largest index such that (i) v_i is the right child of v_{i-1} , (ii) v_{i-1} has the operator $|$, (iii) the max value M at the left child of v_{i-1} is larger than N_1 . If such an i exists then we say that N_1 is dominated by M at depth i in T_{op} . For all numbers in the tree determine if they become dominated, at what depth, and by which other number. The implementation of this step will be discussed in the next section.
3. We now construct a *dominance tree* T_d as follows: Let R be the root node of T_d . For every number at a leaf of T_{op} there will be a node in T_d . Sort the non-dominated numbers in T_{op} in increasing order [27], and make them the children of R in left to right order. Then if N_1 dominates N_2 there is an edge between the nodes N_1 and N_2 in T_d , with N_1 being the parent. The children of an internal node N are sorted first in decreasing order by depth of when they became dominated by N , and then all the numbers with the same depth are sorted in increasing order by their numerical value.
4. Compute the pre-order numbering of T_d [80], and have every node N in T_d except R write its value to $ANS[preorder(N) - 1]$. ■

Lemma 6.4.3: *The above algorithm correctly computes the value of T_{op} .*

Proof: We will prove by induction that the pre-order numbering of T_d yields the value of T_{op} . If T_{op} is of height two, then it is easy to check that the pre-order numbering of T_d gives $val(T_{op})$, and also all of the children of the root of T_d are the leaders of $val(T_{op})$. Inductively we assume that if T_{op} of height D or less, then the pre-order numbering of T_d yields $val(T_{op})$, and also that the children of the root of T_d are the leaders of $val(T_{op})$. Now for the induction step we assume that T_{op} is of height $D + 1$.

Case 1: the operator at the root of T_{op} is concatenate. See Figures 6.2, 6.3 and 6.4. Let T_{d_l} be the dominance tree of the subtree rooted at $left(T_{op})$, and let T_{d_r} be the dominance tree of the subtree rooted at $right(T_{op})$. Let M be the rightmost

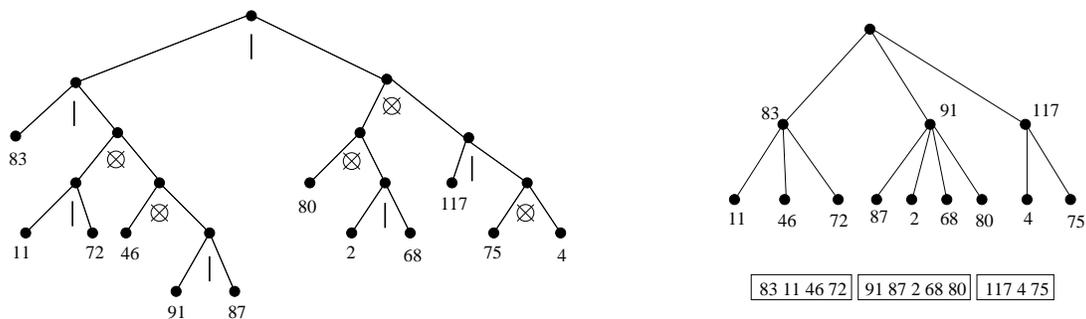


Figure 6.2: Concatenate-merge tree T_L and dominance tree D_L .

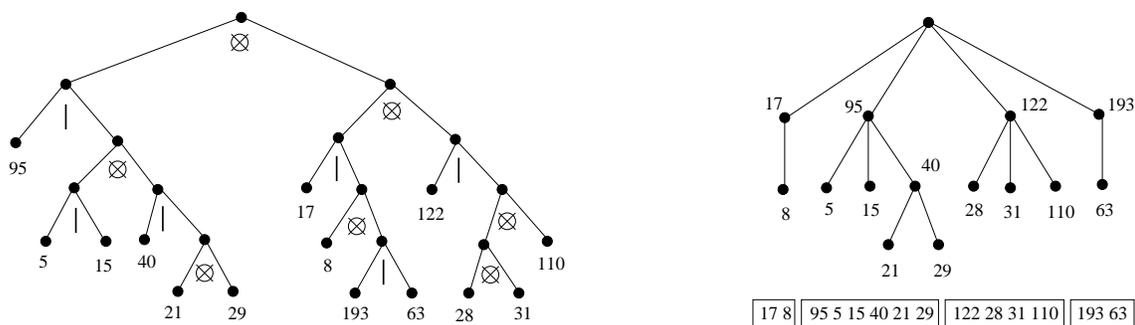


Figure 6.3: Concatenate-merge tree T_R and dominance tree D_R .

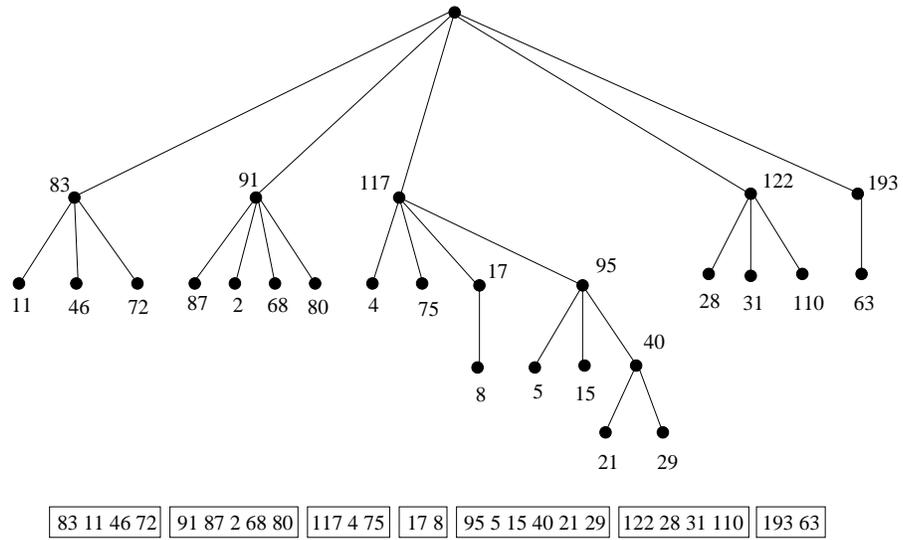


Figure 6.4: Dominance tree for the concatenate-merge tree with $|$ at the root, T_L as the left subtree, and T_R as the right subtree.

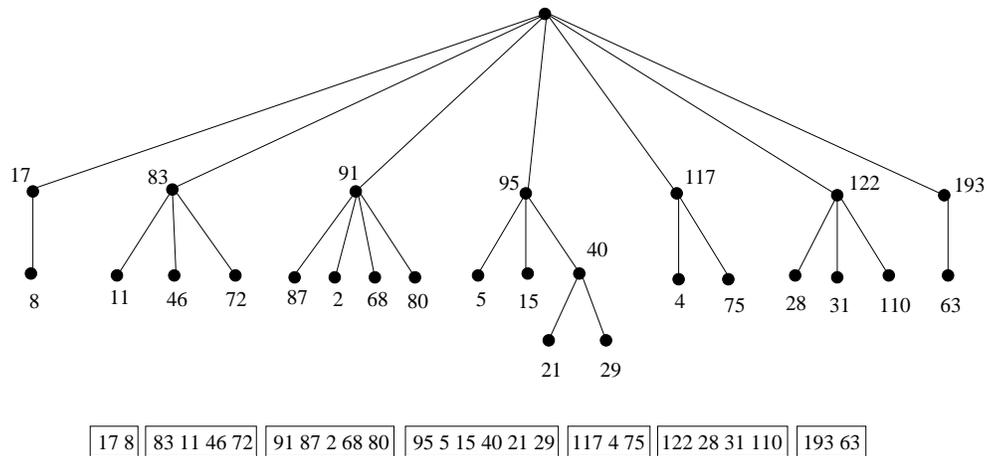


Figure 6.5: Dominance tree for the concatenate-merge tree with \otimes at the root, T_L as the left subtree, and T_R as the right subtree.

child of $root(T_{d_l})$. Based on how the dominance tree is constructed one can see that T_d can be constructed by taking T_{d_l} and then adding the nodes in T_{d_r} as follows: the children of $root(T_{d_r})$ which are less than M are made to point to M (to the right of any elements which already point to M), and the children of $root(T_{d_r})$ which are greater than M are made to point to $root(T_{d_l})$ (to the right of the current children of $root(T_{d_l})$). This is because M is the largest element in $left(T_{op})$, and hence is the only element in $left(T_{op})$ that can dominate an element in $right(T_{op})$. Since from the induction hypothesis we assume that the preorder traversal of T_{d_l} yielded the value of $left(T_{op})$ and the preorder traversal of T_{d_r} yielded the value of $right(T_{op})$ and by the properties of preorder numbering is easy to see that the preorder traversal of T_d yields the value of T_{op} . Also, the property of the children of the root being leaders still holds as all of the nodes hooked up to M were of value smaller than M .

Case 2: the operator at the root of T_{op} is \otimes . See Figures 6.2, 6.3 and 6.5. Define T_{d_l}, T_{d_r} as before. Again, based on the how the dominance tree is constructed one can see that T_d can be constructed by merging the children of T_{d_l} and T_{d_r} based on their value. It is easy to see from the observations made above about computing \otimes and the fact that the children of $root(T_{d_l})$ and $root(T_{d_r})$ are the leaders of $val(left(T_{op}))$ and $val(right(T_{op}))$ and from the properties of preorder numbering that the preorder numbering of T_d yields $val(T_{op})$. Also, the children of the root of T_d are the leaders of the $val(T_{op})$. ■

6.4.3 Solving a Line Segment Query Problem

In order to construct the dominance tree efficiently our method relies on the solution to a query problem involving line segments and rays. The solution utilizes both fractional cascading and segment trees.

We now review the *fractional cascading* technique of Chazelle and Guibas [24]. We are given a directed bounded-degree graph $G = (V, E)$, where each node v in G contains a sorted list $C(v)$. The problem is to construct a data structure so that given a walk (v_1, v_2, \dots, v_s) and an arbitrary element x , one processor can locate x in all of the $C(v_i)$'s quickly. An efficient solution involves constructing an “auxiliary” list $A(v)$ for each $C(v)$ list, such that (i) $C(v) \subseteq A(v)$, (ii) $\sum_{v \in V} |A(v)|$ is $O(\sum_{v \in V} |C(v)|)$, and (iii) given the position of x in $A(v)$ one can locate x in $C(v)$ and in $A(w)$, for each neighbor w of v , in $O(1)$ time. Atallah, Cole and Goodrich [4] derive the following

lemma for this problem:

Lemma 6.4.4: *Given a directed bounded-degree graph $G = (V, E)$, one can build a fractional cascading data structure for G , including all the auxiliary lists $A(v)$ for each $v \in V$, in $O(\log N)$ time using $O(N)$ space with $O(N/\log N)$ processors on a CREW PRAM, where N is $|V| + |E| + \sum_{v \in V} |C(v)|$.*

Now we will review the segment tree, an important data structure in computational geometry [65]. Assume we have t horizontal line segments with endpoints in $[1, \dots, s]$. The segment tree will have $s - 1$ leaves, and the i th leaf in a left to right order on the segment tree corresponds to the interval $[i, i + 1]$. Each internal node corresponds to an interval which is the union of its two children's intervals. Note that we require that the segment tree be near-balanced, i.e. will always have a depth of $O(\log s)$. We say a segment *covers* a node if it spans the interval at that node but not at the node's parent. Notice that each segment can cover up to $O(\log s)$ nodes. The *cover list* of a node consists of all the segments in the input which cover that node.

We are given a set H of s horizontal line segments in the plane with integer x coordinates in the range $[1, \dots, s]$. Let $FunS$ be a one-to-one function from H to the integers. Also, we are given a set R of s query rays, where each ray is of the form $\overrightarrow{(x + .5, y) - (x + .5, -\infty)}$ where x is an integer. Let $FunR$ be a function from R to the integers. Also, we require that the y -coordinates for all of the rays and all of the segments be distinct. In addition, we make the restriction that for every ray $\overrightarrow{(x + .5, y) - (x + .5, -\infty)}$ and line segment $\overline{(a, b) - (c, b)}$ with $a < x + .5 < c$ that $y \geq b$. For every ray r we wish to find the segment s intersected by r with largest y -coordinate such that $FunR(r) \leq FunS(s)$.

- Using the techniques of [4] construct the segment tree T_{seg} on H and at every cover list have the segments sorted in decreasing order by y -coordinate.
- For every cover list $C(v)$ in T_{seg} containing the segments $C(v)_1, C(v)_2, \dots, C(v)_x$ (with segment $C(v)_1$ having the largest y -coordinate) compute $S(v)_i = \max\{FunS(C(v)_j) \mid 1 \leq j \leq i\}$ for all $1 \leq i \leq x$.
- Perform fractional cascading on the $S(v)$ arrays constructed in the last step.

- Consider query ray $r = \overrightarrow{(x + .5, y) - (x + .5, -\infty)}$. We visit every node v in T_{seg} whose interval contains $x + .5$. Clearly r intersects every segment in $S(v)$. Find the rank i of $FunR(r)$ in $S(v)$ such that $S(v)_i < FunR(r) \leq S(v)_{i+1}$. $C(v)_{i+1}$ is the only segment in $C(v)$ which could be the solution for the query ray r . After we visit the $O(\log s)$ nodes whose interval contains $x + .5$ we have $O(\log s)$ segments which intersect r with $FunS$ values larger than $FunR(r)$. The segment with largest y -coordinate is the solution for query ray r . Fractional cascading allows all of the rankings for all of the rays to be performed in $O(\log s)$ time with $O(s)$ processors.

Thus this problem can be solved in $O(\log s)$ time with $O(s)$ processors using $O(s \log s)$ space on a CREW PRAM given $O(s)$ query rays and line segments.

6.4.4 Constructing the Dominance Tree

We now give a procedure for constructing the dominance tree T_d from the concatenate-merge tree T_{op} .

- Compute the maximum value for every internal node of T_{op} with respect to the subtree rooted at the node [55].
- Starting at one, number the leaves in a left to right order [80]. If a leaf is numbered s then assign the leaf the interval $[s, s + 1]$.
- For every internal node of T_{op} compute the union of the intervals in all the leaves below it.
- Let u be an internal node with the concatenate operator at depth D . Let v be its left child and w be its right child. Let $[i_1, i_2]$ be the interval defined by w . Generate all such horizontal line segments $\overline{(i_1, D + 1), (i_2, D + 1)}$ and associate with that segment the maximum value stored at v .

Let l be a leaf node with depth D storing the value e and having interval $[i, i + 1]$. To find the number that dominates e we wish to find the first segment which intersects the ray $\overrightarrow{(i + .5, D), (i + .5, -\infty)}$ which has associated max value greater than e . The algorithm presented above allows this to be solved in $O(\log s)$ time with $O(s)$ processors.

- Let r be a root node, and make its children all of the elements which are not dominated in T_{op} . Sort the tree so that the children of a node are ordered first

by the depth that they become dominated, and then by the weight of the child [27]. The construction of T_d is then complete.

Thus the dominance tree can be constructed with $O(s)$ processors in $O(\log s)$ time using $O(s \log s)$ space on a CREW PRAM. Hence, with our earlier observations we can find the Prim sequence of a tree with $O(n)$ processors and $O(\log n)$ time using $O(n \log n)$ space on a CREW PRAM.

6.4.5 Certifying Minimum Spanning Trees

We are ready to describe the algorithms Mst_1 and Mst_2 for certifying the problem of finding an MST of a graph. These algorithms use some certification trail programs from the previous chapter as subroutines.

Algorithm Mst_1 : Takes as input a graph G . Outputs a MST T of G along with a certification trail.

1. Compute a MST T of G using any MST generation algorithm.
2. Arbitrarily choose a vertex v_0 .
3. Compute the Prim edge sequence $O_{v_0} = (e_1, \dots, e_{n-1})$ and corresponding Prim vertex sequence $O'_{v_0} = (v_1, \dots, v_{n-1})$ of T starting from v_0 .
4. Compute $Cart_1((W(e_1), \dots, W(e_{n-1}))) = (\Upsilon, \Upsilon_{\square})$.
5. Let the edges of G be $e'_i = (v_{l_i}, v_{r_i})$ (with $l_i < r_i$) for $1 \leq i \leq m$. Compute $LcaQ_1(\Upsilon, ((l_1 + 1, r_1), \dots, (l_m + 1, r_m))) = (A, A_{\square})$.
6. Output $(T, (v_0, O_{v_0}, O'_{v_0}, \Upsilon, \Upsilon_{\square}, A, A_{\square}))$.

Algorithm Mst_2 : Takes as input a graph G , T , and a certification trail of the form $(v_0, (e_1, \dots, e_{n-1}), (v_1, \dots, v_{n-1}), \Upsilon, \Upsilon_{\square}, A, A_{\square})$.

1. For every edge $e_i = (v_j, v_k)$ (with $j < k$), check that $k = i$.
2. Check that $Cart_2((W(e_1), \dots, W(e_{n-1})), \Upsilon, \Upsilon_{\square}) = ok$.
3. Let the edges of G be $e'_i = (v_{l_i}, v_{r_i})$ (with $l_i < r_i$) for $1 \leq i \leq m$. Check that $Lca_2(\Upsilon, ((l_1 + 1, r_1), \dots, (l_m + 1, r_m)), A, A_{\square}) = ok$.
4. Let $A = (a_1, a_2, \dots, a_m)$. Check that $W(e_{a_i}) \leq W(e'_i)$ for $1 \leq i \leq m$.

Let T_{mst} , P_{mst} and S_{mst} be functions for the time, processors and space needed by the MST generation algorithm used. By the above analysis we see that Mst_1 can be implemented to run on a CREW PRAM in $O(T_{mst} + \log(n))$ time with $O(P_{mst} + n + m/\log(n))$ processors using $O(n \log n + S_{mst})$ space. Also, Mst_2 runs in constant time with $O(n + m)$ processors on a polling CREW PRAM.

Lemma 6.4.5: *Mst_1 and Mst_2 as described above certify the MST problem.*

Proof: The correctness of Mst_1 and Mst_2 follows from the correctness of the algorithm *RangeCheckMST*. ■

Chapter 7

Formally Verifying Certifier Programs

In this chapter, we demonstrate how the certification-trail technique is applicable to the formal verification process. Traditionally, when formal verification is used to increase software reliability, a program F is developed to solve a problem P and is proven correct. Then, whenever F is given an input and produces an output, one can have great certainty that there is no error in the output due to software faults (assuming that there are no errors in the specification, and that the theorem prover is correct). We propose a different scheme. In the new scheme, when given a problem P , we use the certification-trail technique and develop programs F_1 and F_2 for P . We propose to apply traditional software testing techniques to F_1 , and only prove F_2 is correct. Then to solve P for some particular input, F_1 is invoked to produce the output and the certification trail. If F_2 accepts the output, then since F_2 has been proven correct, we know that it is not possible for the output to be incorrect due to software faults in *either* F_1 *or* F_2 .

Depending upon the particular application, reporting that F_1 has made an error in place of actually producing a correct output may be sufficient. If it is not sufficient, then more steps need to be taken. A hybrid approach is to also develop a program F which solves the problem and prove it correct, and then invoke it when the certifier F_2 detects an error. We will discuss this more in the next section.

7.1 Advantages of Certification Trails

In this section we describe the types of advantages that result from using the certification-trail technique when providing extremely reliable software and hardware systems.

7.1.1 Software Development Advantages

First we discuss some of the potential advantages of using the certification-trail technique in terms of easing the task of providing formally verified software. The first advantage is that proving the certifier correct will often be significantly easier than proving the original program correct. This is because the certifier acts as an answer checker, and hence will often have a significantly simpler structure than the original program which has to actually compute the answer. If it is the case that reporting an error is not sufficient and we must output a correct answer, we can also develop and prove correct a program F to invoke when F_1 fails. While it may appear that we lose

the advantage that F_2 is simpler than F , this is not necessarily the case. Although F_1 is not proven correct, rigorous software testing can give us confidence that it fails only on a very small percentage of inputs. Since presumably F will be run rarely, it would be acceptable from a performance standpoint to deemphasize efficiency issues when coding F , and instead keep F simple so that it will be easy to prove correct.

The second advantage is that a formally verified certifier program can be used with any first phase program, provided that the particular first phase program produces an appropriate certification trail. An example where this is useful is in the case of sorting. Depending upon the data to be sorted, there are a variety of different sorting algorithms which may be the most efficient. For example, quicksort, radix sort, shell sort, and even insertion sort all have inputs for which they individually perform better than the other sorts. In order to take advantage of the most appropriate sort for the data at hand, each individual sort would have to be formally verified to be correct. In addition, each implementation of the sorting program using a different programming language would require a separate proof of correctness. When we present the sorting certifier, we show that it is possible to modify virtually any sorting algorithm to produce an appropriate certification trail.

The third advantage is that the certification-trail technique introduces a clear separation between the tasks of writing an efficient program to solve a problem, and writing a verifiable program to check solution instances to that problem. Thus, it becomes possible to have a team of expert programmers write efficient imperative code to solve the problem, and then have a team of theorem proving experts write a program in a language more suitable for verification that checks the output of the other program. This would lead to an advantageous situation: efficiently executed programs which do not need to be formally verified, and easily verified checkers which do not need to be coded tightly because they inherently have very fast running times. One should still have a high degree of confidence in the correctness of F_1 , or at least believe that F_1 is correct for the vast majority of inputs, but this can be achieved through traditional software testing techniques.

7.1.2 Run-Time Advantages

The use of the certification-trail technique can provide extremely reliable software and hardware systems while greatly diminishing the resources expended at run-time.

In general, there are many steps that can be taken to increase the probability that a correct answer is output from a program. Here are some possibilities:

- Using formally verified software (the associated penalty will be $K_{Software}$).
- Running on formally verified hardware or hardware that is reliable for other reasons ($K_{Hardware}$).
- Using a formally verified compiler ($K_{Compiler}$).
- Using time redundancy (K_{Time}).
- Using simpler (and, hence, more reliable) algorithms ($K_{Algorithm}$).

We will now briefly discuss each of the above steps in terms of its associated penalty. For simplicity, in this section we will consider the penalties in terms of the increased execution time of programs.

In the case of formally verified software, the penalty is due to compromises made in program efficiency in an effort to make formal verification of the programs more tractable. For example, one might choose to program in Prolog or pure Lisp, rather than C, Fortran, or assembly language because formal verification is generally considered easier when using the former languages as opposed to the latter ones. Also, even if one uses an efficiently executed language, one might still avoid writing tightly coded, highly optimized programs because it would make formal verification more difficult. In the case of using formally verified hardware, a penalty results because such hardware is typically going to be simpler and less efficient than non-formally verified hardware. A formally verified compiler is likewise going to be able to apply less extensive optimizations and will likely produce less efficient code. Using time redundancy (which detects transient hardware faults) obviously multiplies the time needed by a factor of n , where n is number of times the program is run. Finally, for many problems there is a choice of algorithms which range from very efficient but very complicated and hard to program correctly, to not very efficient but very simple and easier to program correctly. While it would be desirable to use the complicated algorithms from an efficiency point of view, it may not be justifiable if their complexity makes them unreliable.

For ease of exposition, let us assume that the time penalty associated with each step is simply a constant multiplicative factor. For any problem, let the function

$time()$ give the running time of an idealized baseline program that took none of the above steps mentioned for increased reliability. Thus, $time(P)$ would measure the time that an unverified program solving problem P , running on unverified hardware, and without any other reliability enhancements would require. Thus, if we were to write a program F in Lisp, and then run it on a formally verified processor, the running time of F would be approximated by $time(P) * K_{Software} * K_{Hardware}$, and we would have to compare this to a program solving P that was written in C and was run on an ordinary processor, and hence only took $time(P)$.

So assume that the steps that we will take for increased reliability have been selected, and let K_1, K_2, \dots, K_j be the associated penalty factors. Since the penalties are multiplicative, let $K_{Total} = \prod_{i=1}^j K_i$, and we get a total running time of $time(P) * K_{Total}$. But now consider the case where we develop a certification-trail solution F_1 and F_2 for P . We can think of F_1 as solving a problem P_1 (where P_1 includes problem P but also involves the generation of the certification trail), and F_2 as solving a problem P_2 (where P_2 is the problem of checking the output of P_1). Now, while there may be some advantages to taking all of the reliability measures discussed above when solving P_1 , it is only really necessary to take those measures when solving P_2 . Thus we get a total running time of $time(P_1) + time(P_2) * K_{Total}$. Here we assume that F_1 fails rarely, so that this formula is a close approximation to the formula for expected run time that accounts for the occasional cost of having to run F when F_1 fails. Since the time required to solve P_1 is typically much smaller than that of solving P_2 , significant computational savings are made by only applying the penalties when solving P_2 . Our assumption is that the reliability measures are only going to be taken when solving P_2 . This is perhaps a bit simplistic for some of the proposed reliability measures. For example, is someone likely to acquire special hardware that was very reliable just to run the certifier? But even in this case, one could view the hardware running the certifier as a special type of watchdog monitor processor [63]. It seems quite reasonable to use a watchdog processor that was inherently more reliable than the main processor.

For concreteness, assume that for the problem sizes which we are interested in solving, that $time(P_1)$ is 5% greater than $time(P)$, and that $time(P_2)$ is 10% of $time(P)$. These are very typical of results which we have obtained when applying the certification-trail technique to a range of problems [75][76][84][77]. In Figure 1,

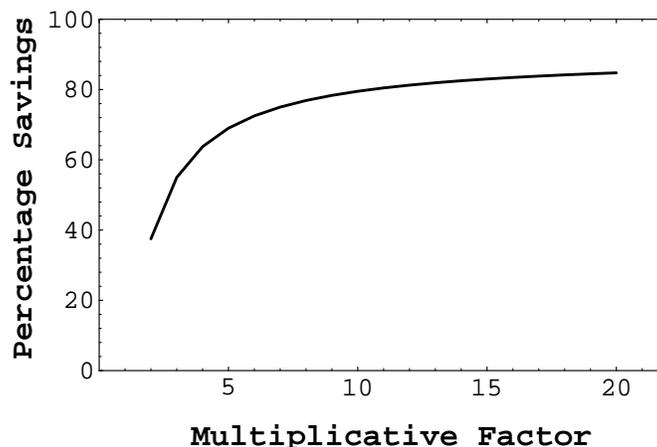


Figure 7.1: Savings possible when using the certification-trail technique.

we present a graph where the x-axis is a range of possible values of K_{Total} from 2 to 20, and the y-axis is the percentage savings resulting from using the certification-trail technique. We define the percentage savings as

$$\%Savings = 100 - 100 * (time(P_1) + time(P_2) * K_{Total}) / (time(P) * K_{Total}).$$

This graph allows us to estimate the savings that we would realize from using the certification-trail technique when solving a given problem. For example, assume that we were interested in the problem P of finding the convex hull of a set of points, and we wanted our software to be formally verified and run on a highly reliable hardware system. Let F be a program that solves P , and let F_1 and F_2 be a certification-trail solution for P . By previous work [77], the assumption that $time(P_1)$ is 10% greater than $time(P)$, and that $time(P_2)$ is 5% of $time(P)$ is actually conservative. Now, if we estimate the value of K_{Total} as 5, then by Figure 1 we see that using the certification-trail technique would yield a total savings of almost 70% compared to traditional approach of running the formally verified F on a highly reliable system.

7.2 The Boyer-Moore Theorem Prover

We now give a brief introduction to the Boyer-Moore Theorem Prover (B&MTP). This description is very similar to the one that appears in [67]. For more details the reader is advised to consult the books [18][19]. For a survey of the major accomplishments using

the B&MTP the reader can consult [53]. The Boyer-Moore Logic is a quantifier-free first-order logic with equality. All variables are assumed to be universally quantified. The language is closely related to pure Lisp and consists of variables and functions combined in a prefix notation. A constant in the logic is a function which is given no arguments.

The basic function symbols in the logic are TRUE, FALSE, IF and EQUAL. The first two are functions of zero arguments and are to be viewed as distinct constants. The function IF takes three arguments, e.g. X, Y, and Z, and if X is TRUE returns Y, and otherwise returns Z. The function EQUAL takes two arguments, e.g. X and Y, and returns TRUE if X is equivalent to Y, and FALSE otherwise.

A theory of the natural numbers is built into the prover. The logic also contains a definition principle which can be used to introduce new function definitions. The B&MTP will accept the definition of a function only if it can show that the function must terminate given any input. Also, the B&MTP allows a function to be defined only in terms of previously defined functions.

The inference rules used by the B&MTP are as follows:

- Equality reasoning, that permits the substitution of equals for equals in all expressions.
- The rule of instantiation, where an instance of a theorem is a theorem. An instance of a theorem is generated by substituting uniformly the same expression for all occurrences of a variable.
- Noetherian induction. The induction rule forms the backbone of the B&MTP. Any function definition admitted by the system yields a valid induction schema. The Noetherian induction principle lets the B&MTP make use of these induction schemas to prove new conjectures.

The B&MTP has a hint facility by which a user can instruct the prover to use an existing function definition as an induction scheme. Also, the user can instruct the prover in which previously proved theorems are important, and which can be ignored. Although an extension to the B&MTP has been developed where the user can guide the prover through very specific steps while constructing the proof, for the most part the proofs we present in this chapter are automatic. That is, after a few relevant hints have been made, the B&MTP is able to prove the theorem with no additional user intervention. Of course, the intermediate theorems which we chose to

prove and the order in which we proved them were critical in enabling the B&MTP to prove the final theorems that we were interested in.

7.3 Definition of the Sorting Certifier

In this section we present a certifier program for sorting. In an effort to present a program that is as general as possible, we do not define our certifier program in terms of any fixed comparison operation. Instead, we introduce a function symbol *ourlessp*, and constrain it to have only those order properties that we need to prove the soundness and completeness of the certifier. The order properties that we need are: 1) *ourlessp* is transitive, and 2) *ourlessp* is anti-symmetric and irreflexive.

CONSERVATIVE AXIOM: *ourlessp*-constrain
 $((\text{ourlessp}(a, b) \wedge \text{ourlessp}(b, c)) \rightarrow \text{ourlessp}(a, c))$
 $\wedge (\text{ourlessp}(a, b) \rightarrow (\neg \text{ourlessp}(b, a)))$

Now we give a formal definition for what it means for a list *o* to be the list *i* sorted. Note that this is not as simple as checking that the elements in *o* are non-decreasing. It is also necessary to check that the elements in *o* are a permutation of the elements in *i*. Function *remove1*(*a*, *x*) removes the first copy of element *a* from the list *x*. If *a* does not appear in *x*, then the list *x* is returned unmodified. We say that a list *x* is a subbag of a list *y*, if every element in *x* appears at least as many times in *y*. A bag is a set which allows for repeated elements. Predicate *subbagp*(*x*, *y*) performs this check on its arguments. Predicate *permutationp*(*x*, *y*) checks the permutation property by testing that *x* is a subbag of *y*, and also that *y* is a subbag of *x*. Predicate *ordered*(*o*) checks that *o* is a non-decreasing sequence. Finally, we define *sort*(*i*, *o*) which checks that the list *o* is the list *i* sorted. The definitions for *ordered*, *subbagp*, and *permutationp* are standard definitions used by many members of the theorem proving community.

DEFINITION:
`remove1(a, x)`
`= if listp(x)`
`then if car(x) = a then cdr(x)`
`else cons(car(x), remove1(a, cdr(x))) endif`
`else x endif`

DEFINITION:

```

subbagp(x, y)
= if listp(x) then (car(x) ∈ y)
     $\wedge$  subbagp(cdr(x), remove1(car(x), y))
  else t endif

```

DEFINITION: permutationp(x, y) = (subbagp(x, y) \wedge subbagp(y, x))

DEFINITION:

```

ordered(o)
= if (o  $\simeq$  nil)  $\vee$  (cdr(o)  $\simeq$  nil) then t
  else ( $\neg$  ourlessp(cadr(o), car(o)))  $\wedge$  ordered(cdr(o)) endif

```

DEFINITION: sort(i, o) = (ordered(o) \wedge permutationp(o, i))

The definition of *ordered* contains the symbol \simeq . The boolean expression (*expr* \simeq **nil**) is true if *expr* is an “atom.” An atom is any data type which is not a list, such as an integer or a symbol. The boolean expression (*expr* = **nil**) is true if *expr* is the specific symbol **nil**. While it is clear that the predicate *sort*(*i*, *o*) captures precisely what we mean for the list *o* to be the list *i* sorted, *sort*(*i*, *o*) is not executed efficiently as a Lisp program. This is because of the predicate *subbagp*(*x*, *y*), which can require $\Theta(\text{length}(x)\text{length}(y))$ to run in the worst case. Thus we need a more clever way to check that a list is the sorted version of another list.

This will be accomplished by using a certification trail. Intuitively, the certification trail will consist of a permutation that when applied to the output, yields the input [75]. The certification trail will be constructed during the first phase by tagging each element of the input with its index position. Then after sorting the numbers in the input, the tags will give the desired permutation. It is easy to modify most sorting algorithms to also keep track of the tags. A similar idea for checking sorting also appears in [69]. However, since in their model they treat the sorting program as a black box which can not be modified, they are forced to encode the tags as part of the input. This black box restriction gives their method a desirable generality; however, it obviously causes problems when the input numbers are from a finite domain, such as for most number representations on computers today. While this difference might be viewed as somewhat of a technicality for the problem of sorting, the ability to modify the program computing the output so that it explicitly constructs a certification trail is precisely what gives the certification-trail

technique an advantage over the probabilistic program checking model [14][69] for many problems.

We now present the routines *make-array*, *access* and *assign* which operate on arrays. Our intention here is to model the notion of an array very closely, so that in the actual program implementation we can replace calls to these functions with calls to built-in array routines to insure a fast running time. We also introduce a 0-ary function symbol SYMBOL, which is meant to be a special symbol that does not appear in the input or in the output being checked.

DEFINITION:

```
make-array (j)
= if (j  $\notin$   $\mathbf{N}$ )  $\vee$  (j = 0) then nil
  else cons (SYMBOL, make-array (j - 1)) endif
```

DEFINITION:

```
access (a, j)
= if (j  $\notin$   $\mathbf{N}$ )  $\vee$  (a  $\simeq$  nil) then SYMBOL
  elseif j = 0 then car (a)
  else access (cdr (a), j - 1) endif
```

DEFINITION:

```
assign (a, j, v)
= if (j  $\notin$   $\mathbf{N}$ )  $\vee$  (a  $\simeq$  nil) then a
  elseif j = 0 then cons (v, cdr (a))
  else cons (car (a), assign (cdr (a), j - 1, v)) endif
```

We are now ready to begin defining our sorting certifier. The input to our certifier, which we call *certsort*, consists of the input list *i*, the output list *o*, and a certification trail list *c*. The certification trail *c* is meant to be a permutation that when applied to *o* yields *i*, and is what enables *certsort* to check that *o* is a permutation of *i* quickly. Before defining *certsort* we need to define some auxiliary functions. Function *construct*(*a*, *o*, *c*) modifies the array *a* according to the list of elements *o* and indices *c*. The *j*th element in *o* is assigned into *a* as specified by the *j*th index in *c*. Function *construct*(*a*, *o*, *c*) is initially called with each entry of *a* containing the special value SYMBOL, and the function *doconstruct*(*o*, *c*) provides a clean interface to do this.

DEFINITION:

```
construct(a, o, c)
= if (o  $\simeq$  nil)  $\vee$  (c  $\simeq$  nil) then a
  else construct(assign(a, car(c), car(o)), cdr(o), cdr(c)) endif
```

DEFINITION:

```
doconstruct(o, c) = construct(make-array(length(o)), o, c)
```

It is useful to take the array returned by *doconstruct* and make a list by extracting elements in the array. The function *contents*(*a*, *j*, *k*) extracts the elements in the array *a* between (and including) the indices *j* and *k*, and returns them in a list. There is a technical check in the body of *contents*(*a*, *j*, *k*) which checks that each element in *a* in the specified range does not equal SYMBOL. All elements that equal SYMBOL are not included in the output of *contents*. Although a correct program would result if *contents* returned all of the elements in the specified range, the version presented here is far more useful for reasoning about the execution of *construct*.

DEFINITION:

```
contents(a, j, k)
= if (j  $\notin$   $\mathbf{N}$ )  $\vee$  (k  $\notin$   $\mathbf{N}$ )  $\vee$  (k < j) then nil
  elseif access(a, j)  $\neq$  SYMBOL
  then cons(access(a, j), contents(a, 1 + j, k))
  else contents(a, 1 + j, k) endif
```

DEFINITION: allcontents(*a*) = contents(*a*, 0, length(*a*) - 1)

Before we define *certsort*, we define the predicate *slistp*. Predicate *slistp*(*x*) returns *T* if either *x* is **nil**, or *x* is a **nil** terminated list. This is a common well-formedness property required of the inputs to a Lisp program. Afterwards, we define the predicate *certsort*(*i*, *o*, *c*). Predicate *certsort*(*i*, *o*, *c*) calls the function *doconstruct*(*o*, *c*) which constructs an array from *o* and *c*. The critical property of *doconstruct*(*o*, *c*) is that a list of the elements in the array returned is a subbag of *o*. If the input list *i* equals the list *allcontents*(*doconstruct*(*o*, *c*)), we have that *i* and *o* are permutations of each other. We will explain the reason for this more in the last section of this chapter.

DEFINITION:

```
slistp(x)
= if x  $\simeq$  nil then x = nil
  else slistp(cdr(x)) endif
```

DEFINITION:

```

certsort (i, o, c)
= ((length (i) = length (o))
  ∧ (SYMBOL ∉ i)
  ∧ (SYMBOL ∉ o)
  ∧ slistp (i)
  ∧ slistp (o)
  ∧ ordered (o)
  ∧ (allcontents (doconstruct (o, c)) = i))

```

Using the Boyer-Moore Theorem Prover, we proved that *certsort* is sound. That is, if *certsort* returns true given an input/output pair and some certification trail, then the input/output pair really is correct. However, this only establishes half of the formal definition of the certification-trail method. We still need to give a function F_1 which if given an *i*, produces an *o* and a *c* such that $\text{certsort}(i, o, c) = T$ holds. In the last section of this chapter, we provide most of the details for both the soundness proof, and also for how to construct the function F_1 . For now, we will only state the soundness theorem that we proved.

THEOREM: certsort-sort

```

certsort (i, o, c) → sort (i, o)

```

7.4 Experimental Results

In this section we discuss our experimental results. We performed some minor modifications on procedure *certsort* so that it ran more quickly. First, we defined the predicate *test-memberlength*(*i*,*o*) which tests that *i* and *o* are of the same length and that neither contain SYMBOL. Also, we defined the predicate *test-equal*(*a*,*i*,*j*) which if called initially with *j* = 0, tests that the contents of the array *a* are equal to the contents of the list *i*. We now give their formal definitions.

DEFINITION:

```

test-memberlength (i, o)
= if (i ≈ nil) ∨ (o ≈ nil) then (i = nil) ∧ (o = nil)
  elseif (car (i) = SYMBOL) ∨ (car (o) = SYMBOL) then f
  else test-memberlength (cdr (i), cdr (o)) endif

```

DEFINITION:

```
test-equal(a, i, j)
= if i  $\simeq$  nil then length(a) = j
  elseif access(a, j) = car(i) then test-equal(a, cdr(i), 1 + j)
  else f endif
```

DEFINITION:

```
faster-certsort(i, o, c)
= (test-memberlength(i, o)
    $\wedge$  ordered(o)
    $\wedge$  test-equal(doconstruct(o, c), i, 0))
```

We have proven that the modified versions are both sound and complete. The proofs for the modified versions were not difficult, and we do not discuss them in this paper. Next, we had to perform some minor modifications to the programs so they would conform to Common Lisp conventions. For example, the predicate in the Boyer-Moore logic to test if an object is not a cons cell is *nlistp*, and in Common Lisp the predicate is *atom*. Also, we modified *faster-certsort* and *ordered* so that the comparator function is passed in and then applied to the data using the Lisp *funcall* routine. This modification makes it easier to use *faster-certsort* with different types of data, and also allows for a more fair comparison to the built in *Sort* routine which also takes a comparator function as an input parameter. We replaced all occurrences of SYMBOL with the constant -1 .

Our most major modification was to use the built in array routines for Allegro Common Lisp, instead of the ones we defined earlier. There is a difficulty here. Since the Lisp provided by the B&MTP is purely functional, all the array routines that we defined were “non-destructive.” That is, when an *assign* is performed, the array is copied, and the assignment only takes places in the new copy. For obvious reasons of efficiency, the array routines built into Allegro Common Lisp are “destructive,” meaning that there is only one copy of an array, and all modifications take place on the one copy. Resolving this problem completely in the B&MTP is difficult, and would take us far afield from our main goal. Instead of this approach, we have constructed a short hand-proof that the meaning of our programs is the same regardless of which type of arrays are used. In addition, there are numerous papers in the area of Programming Language Theory which deal with the issue of efficiently implementing arrays and other constructs in functional languages [45][47][70]. Using

these and other techniques, a sophisticated compiler should be able to perform these optimizations automatically.

For completeness, we now give the final Common Lisp version of our certifier. Throughout this paper we used a pretty printer to format the function definitions and theorem statements. Though this code appears very different from the definitions of *certsort* and *faster-certsort*, it is actually almost identical to the programs we reasoned about using the B&MTP.

```
(defun mymake-array (j)
  (make-array j :initial-element -1))
; "make-array" is the Common Lisp routine for allocating
;   a new array

(defun assign (A j v)
  (declare (type (array t 1) A))
  (if (and (numberp j)
           (<= 0 j)
           (< j (length A)))
      (setf (aref A j) v) )
  A )
; "aref" returns a pointer to the j'th element of A
; "setf" then assigns that location of A to contain v

(defun access (A j)
  (declare (type (array t 1) A))
  (if (and (numberp j)
           (<= 0 j)
           (< j (length A)))
      (aref A j)
      -1 ))

(defun ordered (O CMP)
  (if (or (atom O) (atom (cdr O)))
      T
      (and (not (funcall CMP (cadr O) (car O)))
            (ordered (cdr O) CMP) )))

(defun construct (A O C)
```

```

(if (or (atom O) (atom C))
    A
    (construct (assign A (car c) (car o)) (cdr O) (cdr C)) ))

(defun permute (O C)
  (construct (mymake-array (length O)) O C))

(defun test-equal (A I j)
  (if (atom I)
      (equal (length A) j)
      (if (equal (access A j) (car I))
          (test-equal A (cdr I) (+ j 1))
          nil )))

(defun test-memberlength (I O)
  (if (or (atom I) (atom O))
      (and (equal I nil) (equal O nil))
      (if (or (equal (car I) -1) (equal (car O) -1))
          nil
          (test-memberlength (cdr I) (cdr O))))))

(defun faster-certsort (I O C CMP)
  (and (test-memberlength I O)
       (ordered O CMP)
       (test-equal (permute O C) I O)))

```

Finally, we discuss our timing methodology. Our tests were performed on a Sun Sparc II with 16 megabytes of main memory. The Lisp compiler used was Allegro Common Lisp version 4.2.beta. All the programs were compiled for maximum speed. All of the basic input lists were randomly generated using the built in function *random*, and consisted of integers in the range $1, \dots, 10 * n$, where n was the length of the list being generated. The comparison function used was the built in “<” operator. Different randomized inputs were used for *C-Sort* and *faster-certsort*. An appropriate certification trail for *faster-certsort* was generated using the method presented in the appendix. We compared the running time of *faster-certsort* to the built in sorter in Allegro called *Sort*. We believe that it is safe to assume that a sorting program written in Common Lisp itself would not be significantly faster than the built in sort function,

Input Size	Allegro-Sort	C-Sort	Faster-CertSort	% Savings
5000	.338	.112	.0951	38.7
10000	.706	.248	.183	39
50000	4.06	1.54	.921	39.4
100000	8.65	3.33	1.84	40.2
200000	18.09	7.17	3.71	39.9
300000	31.16	11.18	5.71	45.8

Table 7.1: Execution speed comparisons for Faster-CertSort, Allegro-Sort, and C-Sort.

and hence that the timing data here gives a fair picture of the time savings possible from using our approach. The same inputs were used with both *faster-certsort* and the built in sort routine. We tested each program on 10 inputs for each problem size, and averaged the results.

Table 7.1 summarizes our experimental results. The times for Allegro-Sort, Faster-CertSort and C-Sort are in seconds. C-Sort is a C program that sorts its input and also generates a certification trail. It uses the built in Berkeley Unix *qsort* sorting routine. The column for C-Sort includes the time for generating a certification trail. We compared the cost of running *C-Sort* and *faster-certsort* to the cost of running *Allegro-Sort*. The column for percentage savings is

$$\%Savings = 100 - 100 * (CSort + FasterCertsort)/(AllegroSort)$$

We should point out that efficient sorting programs have been formally verified using the B&MTP. For example, a 68020 version of quicksort was formally verified by Dr. Yuan Yu as part of his dissertation research [20]. However, this was a very difficult project: the modeling of the behavior of the 68020 alone was very hard. An experienced member of the theorem proving community might comment that our proofs are rather straightforward. Far from taking this as a criticism, we would emphasize that this is one of the advantages of our approach. Our program *faster-certsort*, although not difficult to formally verify, can be used with virtually any sorting program, whether the program was written in C, Fortran, assembly language.

7.5 Proofs of Soundness and Completeness

7.5.1 Proof That *certsort* is Sound

Now, we turn our attention to proving that *certsort* is sound. That is, if *certsort*(*i*, *o*, *c*) is true, then *sort*(*i*, *o*) is true as well. This section consists of a sequence of statements of theorems. In most cases, the B&MTP needs only to be advised which of the theorems previously shown are useful in order to automatically prove the current theorem. In some cases, as noted in the text, several additional intermediate theorems need to be derived before the proof of the current theorem can be attempted.

Below we state two important properties of the functions *access* and *assign*. Theorem *r1-model* states that if the *j*th location of an array is assigned a value, then if the *j*th location is accessed that value is retrieved. Theorem *r2-model* states that assigning the *j*th location of an array does not change the value of the *k*th location, provided that $j \neq k$ holds. The B&MTP is able to prove both of these theorems with no assistance.

THEOREM: r1-model

$$((j \in \mathbf{N}) \wedge (j < \text{length}(a))) \rightarrow (\text{access}(\text{assign}(a, j, v), j) = v)$$

THEOREM: r2-model

$$(j \neq k) \rightarrow (\text{access}(\text{assign}(a, j, v), k) = \text{access}(a, k))$$

Although we do not give the proof of them here, the following theorems are critical. The most important of them is *length-subbag-permutationp*. This states that if *x* is a sub-bag of *y*, and both *x* and *y* have the same length, then *x* and *y* must be permutations of each other. This is the very heart of why the checker is sound. Function *doconstruct*(*o*, *c*) returns an array, call it *a*, such that *allcontents*(*a*) is a sub-bag of *o*. Now if *allcontents*(*a*) = *i*, we have that *i* is a sub-bag of *o*. Since *o* and *i* must be of the same length if *certsort*(*i*, *o*, *c*) returns true, by the theorem *length-subbag-permutationp* *i* and *o* must be permutations of each other.

THEOREM: subbag-transitive

$$(\text{subbagp}(x, y) \wedge \text{subbagp}(y, z)) \rightarrow \text{subbagp}(x, z)$$

THEOREM: subbag-self

$$\text{subbagp}(x, x)$$

THEOREM: subbagp-multi
 $(\text{subbagp}(a, \text{append}(ro, b)) \wedge \text{subbagp}(b, \text{cons}(fo, c)))$
 $\rightarrow \text{subbagp}(a, \text{cons}(fo, \text{append}(ro, c)))$

THEOREM: length-subbagp-permutationp
 $((\text{length}(x) = \text{length}(y)) \wedge \text{subbagp}(y, x)) \rightarrow \text{permutationp}(x, y)$

The critical property to show for soundness is that $\text{allcontents}(\text{doconstruct}(o, c))$ is a sub-bag of o . In order to prove this we must use induction, and it becomes necessary to generalize the statement. Thus, instead we prove that

$$\text{subbagp}(\text{contents}(\text{construct}(a, o, c), \theta, \text{length}(a) - 1),$$

$$\text{append}(o, \text{contents}(a, \theta, \text{length}(a) - 1)))$$

This new property can be stated as follows: if we are given an array a which contains a bag of elements s , and we assign the elements in the set o to various locations in a , then the elements in the resulting array are a sub-bag of $s \cup o$. Before we can prove this generalized statement, we need to codify some important properties of the functions contents and construct . The first two theorems are trivial and need no explanation.

THEOREM: contents-nil
 $\text{contents}(\text{make-array}(l), j, k) = \mathbf{nil}$

THEOREM: contents-lessp
 $(k < j) \rightarrow (\text{contents}(a, j, k) = \mathbf{nil})$

The theorem contents-append1 is useful because it shows that $\text{contents}(a, j, k)$ can be found by applying contents to two contiguous portions of a and appending the results. The formulation of this idea in contents-append2 is especially useful. The theorems contents-assign1 and contents-assign2 describe some properties of contents when it is applied to an array that has just been assigned a value. The theorem contents-assign1 states that the contents of an array for a specified range do not change if an assignment is performed on the array at an index that lies outside of the specified range. The theorem contents-assign2 states that if an array is assigned some value at index j , then the contents of the array from j to j is simply a list containing that value.

THEOREM: contents-append1

$$((k \not\prec j) \wedge (l \not\prec k) \wedge (l \in \mathbf{N}) \wedge (j \in \mathbf{N}) \wedge (k \in \mathbf{N})) \\ \rightarrow (\text{contents}(a, j, l) = \text{append}(\text{contents}(a, j, k), \text{contents}(a, 1 + k, l)))$$

THEOREM: contents-append2

$$((k \not\prec j) \wedge (l \not\prec k) \wedge (l \in \mathbf{N}) \wedge (j \in \mathbf{N}) \wedge (k \neq 0) \wedge (k \in \mathbf{N})) \\ \rightarrow (\text{contents}(a, j, l) \\ = \text{append}(\text{contents}(a, j, k - 1), \\ \text{append}(\text{contents}(a, k, k), \text{contents}(a, 1 + k, l))))$$

THEOREM: contents-assign1

$$(((l < j) \vee (k < l)) \wedge (l \in \mathbf{N}) \wedge (j \in \mathbf{N}) \wedge (k \in \mathbf{N})) \\ \rightarrow (\text{contents}(\text{assign}(a, l, \text{val}), j, k) = \text{contents}(a, j, k))$$

THEOREM: contents-assign2

$$((j \in \mathbf{N}) \wedge (j < \text{length}(\text{assign}(a, j, \text{val}))) \wedge (\text{val} \neq \text{SYMBOL})) \\ \rightarrow (\text{contents}(\text{assign}(a, j, \text{val}), j, j) = \text{list}(\text{val}))$$

The theorem *contents-construct* can be proved using induction along with judicious applications of the theorems we have shown. This is the most technical part of the proof of the soundness of *certsort*. In the actual proof, several intermediate helper theorems were shown before the proof of *contents-construct* was attempted.

THEOREM: contents-construct

$$(\text{SYMBOL} \notin o) \\ \rightarrow \text{subbagg}(\text{contents}(\text{construct}(a, o, c), 0, \text{length}(a) - 1), \\ \text{append}(o, \text{contents}(a, 0, \text{length}(a) - 1)))$$

The theorem *contents-construct* can almost be used to prove what we wanted initially. First however, we need to prove that the array returned by *construct* has the same length as the array originally passed to *construct*.

THEOREM: construct-length

$$\text{length}(\text{construct}(a, o, c)) = \text{length}(a)$$

THEOREM: allcontents-doconstruct

$$(\text{SYMBOL} \notin o) \rightarrow \text{subbagg}(\text{allcontents}(\text{doconstruct}(o, c)), o)$$

The next three theorems follow immediately from the work we have already done. Theorem *certsort-sort* is the final statement of soundness of *certsort*.

THEOREM: certsord-ordered
 $\text{certsort}(i, o, c) \rightarrow \text{ordered}(o)$

THEOREM: certsord-perm
 $\text{certsort}(i, o, c) \rightarrow \text{permutationp}(o, i)$

THEOREM: certsord-sort
 $\text{certsort}(i, o, c) \rightarrow \text{sort}(i, o)$

7.5.2 Completeness Result for *certsort*

In the previous section we proved that *certsort* is sound. But this establishes only half of the formal definition for certification trails. We still need to present a function F_1 which if given an i , produces an o and a c such that $\text{certsort}(i, o, c)$ is true.

It is the case that if the input i is unique (i.e. no element appears twice in i) then there is only one certification trail c that satisfies *certsort*. But if at least one element in i appears multiple times, then there are multiple certification trails which satisfy *certsort*. Consider the following i and o , in which the number 16 appears twice:

$$i = [16, 7, 17, 16, 5]$$

$$o = [5, 7, 16, 16, 17]$$

Then both c_1 and c_2 given below are certification trails which satisfy predicate *certsort*. Recall that our arrays start with the index 0.

$$c_1 = [4, 1, 0, 3, 2]$$

$$c_2 = [4, 1, 3, 0, 2]$$

Next we describe a general method for taking a list i and producing the output and an appropriate certification trail. We use the notation $x[j]$ to refer to the j th element in list x . First, construct the list i' of ordered pairs, so that $i'[j] = (i[j], j)$. Next, sort i' to produce o' . For this sort, any comparison function on ordered pairs is acceptable, provided that if $a_1 < a_2$ then $(a_1, j_1) < (a_2, j_2)$ holds. Each different ordering of the pairs which satisfies this constraint produces a different valid certification trail. Construct the output o such that $o[j] = \text{first-component}(o'[j])$. It is clear that o must be ordered. Next construct the certification trail c such that

$c[j] = \text{second-component}(o'[j])$. As an example, let i be as defined above, and let us compare the pairs in i' in the usual lexicographic manner. Thus we get:

$$\begin{aligned} i &= [16, 7, 17, 16, 5] \\ i' &= [(16, 0), (7, 1), (17, 2), (16, 3), (5, 4)] \\ o' &= [(5, 4), (7, 1), (16, 0), (16, 3), (17, 2)] \\ o &= [5, 7, 16, 16, 17] \\ c &= [4, 1, 0, 3, 2] \end{aligned}$$

Analyzing the above example, we see that the critical relationship between i , o , and c is that for every j , $k = c[j]$ is an index such that $i[k] = o[j]$, and also that no two indices in c are the same. Next we proceed to formally present and prove correct the above scheme. We do not give a complete proof here, but instead present most of the interesting theorems which were needed to complete the proof. Function $\text{first}(x)$ is given a list of pairs, and returns a list consisting of the first element of each pair. Function $\text{second}(x)$ is given a list of pairs, and returns a list consisting of the second element of each pair.

DEFINITION:

```
first(x)
= if x ≈ nil then nil
  else cons(caar(x), first(cdr(x))) endif
```

DEFINITION:

```
second(x)
= if x ≈ nil then nil
  else cons(cadar(x), second(cdr(x))) endif
```

Function $\text{pair}(i, j)$ is given a list i , and an index j , and produces a list of pairs i' constructed from i , such that $i'[k] = (i[k], k + j)$. Predicate $\text{pairp}(x)$ returns true if each element in the list x is an ordered pair. Predicate $\text{pordered}(x)$ returns true if the list of pairs x is ordered by first component.

DEFINITION:

```
pair(i, j)
= if i ≈ nil then nil
  else cons(list(car(i), j), pair(cdr(i), 1 + j)) endif
```

DEFINITION:

```
pairp (x)
= if  $x \simeq \mathbf{nil}$  then t
  else listp (cдар (x))  $\wedge$  pairp (cdr (x)) endif
```

DEFINITION:

```
pordered (x)
= if ( $x \simeq \mathbf{nil}$ )  $\vee$  (cdr (x)  $\simeq \mathbf{nil}$ ) then t
  else ( $\neg$  ourlessp (caadr (x), caar (x)))  $\wedge$  pordered (cdr (x)) endif
```

We are now ready to describe the major theorem of this section. We intend to prove that if oo is a permutation of $pair(i, \theta)$, and that $pordered(oo)$ is true, then $certsort(i, first(oo), second(oo))$ is true also. The basic idea of the proof is as follows. First we give a relaxed notion of equality for two lists by defining the predicate $equalish(x, y)$ which returns true if for all j , either $x[j] = y[j]$, or $x[j] = \text{SYMBOL}$, or $y[j] = \text{SYMBOL}$. We then define the function $numnotsymbol(x)$ which given a list x , counts the number of elements of x which are not equal to SYMBOL . The theorem $equalish-equal$ states the fact that if neither list x nor list y contain SYMBOL , yet $equalish(x, y) = T$, then x must indeed equal y . After this theorem has been shown, we will prove the theorem $equalish-construct$ which establishes that the list $doconstruct(first(oo), second(oo))$ is $equalish$ to the original input i , and the theorem $numnotsymbol-construct$ which establishes that $doconstruct(first(oo), second(oo))$ does not contain the element SYMBOL . In terms of the critical relationship between i , o , and c mentioned above, $equalish-construct$ should be thought of as establishing that for all j , $k = c[j]$ is an index such that $i[k] = o[j]$, and $numnotsymbol-construct$ as establishing that no index in c appears more than once. The desired result follows soon after this.

DEFINITION:

```
equalish (x, y)
= if ( $x \simeq \mathbf{nil}$ )  $\vee$  ( $y \simeq \mathbf{nil}$ ) then t
  else ((car (x) = car (y))
         $\vee$  (car (x) = SYMBOL)
         $\vee$  (car (y) = SYMBOL))
     $\wedge$  equalish (cdr (x), cdr (y)) endif
```

DEFINITION:

```
numnotsymbol (x)
```

```

= if  $x \simeq \mathbf{nil}$  then 0
  elseif  $\text{car}(x) = \text{SYMBOL}$  then  $\text{numnotsymbol}(\text{cdr}(x))$ 
  else  $1 + \text{numnotsymbol}(\text{cdr}(x))$  endif

```

THEOREM: equalish-equal

```

(equalish( $x$ ,  $y$ )
 $\wedge$   $\text{slistp}(x)$ 
 $\wedge$   $\text{slistp}(y)$ 
 $\wedge$  ( $\text{length}(x) = \text{length}(y)$ )
 $\wedge$  ( $\text{numnotsymbol}(y) = \text{length}(y)$ )
 $\wedge$  ( $\text{numnotsymbol}(x) = \text{length}(x)$ ))
 $\rightarrow$  ( $(x = y) = \mathbf{t}$ )

```

The proof that $\text{equalish}(\text{doconstruct}(\text{first}(oo), \text{second}(oo)), i) = T$ requires some new definitions and theorems. The predicate $\text{inrange}(x, j)$ returns true if each index in x is valid for use on an array of length j . The theorem *inrange-second* establishes that the indices in $\text{second}(oo)$ are indeed valid. The theorem *access-subbagp-second* is a phrasing of the first part of the critical property that is needed between i , $\text{first}(oo)$, and $\text{second}(oo)$ as was discussed before.

DEFINITION:

```

inrange( $x$ ,  $j$ )
= if  $x \simeq \mathbf{nil}$  then  $\mathbf{t}$ 
  else ( $\text{car}(x) \in \mathbf{N}$ )  $\wedge$  ( $\text{car}(x) < j$ )  $\wedge$   $\text{inrange}(\text{cdr}(x), j)$  endif

```

THEOREM: inrange-second

```

subbagp( $oo$ ,  $\text{pair}(i, 0)$ )  $\rightarrow$   $\text{inrange}(\text{second}(oo), \text{length}(i))$ 

```

THEOREM: access-subbagp-second

```

(subbagp( $oo$ ,  $\text{pair}(i, 0)$ )  $\wedge$   $\text{listp}(oo)$ )
 $\rightarrow$  ( $\text{access}(i, \text{car}(\text{second}(oo))) = \text{car}(\text{first}(oo))$ )

```

The predicate $\text{unique}(x)$ returns true if every element in x appears only once. The theorem *unique-subbagp-second* states that the list $\text{second}(oo)$ is *unique*. This theorem is more difficult to prove than one would expect at first glance. Although we do not go into details, the proof went along the following lines: 1) we showed that if x is a sub-bag of y , and y is unique, then x must be unique, 2) that if x is a sub-bag of y , then $\text{second}(x)$ is a sub-bag of $\text{second}(y)$, and finally 3) $\text{unique}(\text{second}(\text{pair}(i, 0))) = T$ holds. It is the second step that is difficult, and we

actually proved a more general statement that would also yield under the appropriate hypothesis that $first(x)$ is a sub-bag of $first(y)$, or indeed that when any “selector” function is applied to each element of the lists x and y , then the list resulting from applying the selector to x is a sub-bag of the list resulting from applying the selector to y .

DEFINITION:

unique(x)
 = **if** $x \simeq \mathbf{nil}$ **then t**
 else $(\text{car}(x) \notin \text{cdr}(x)) \wedge \text{unique}(\text{cdr}(x))$ **endif**

THEOREM: unique-subbagp-second
 $\text{subbagp}(oo, \text{pair}(i, 0)) \rightarrow \text{unique}(\text{second}(oo))$

Proving that $second(oo)$ is unique has the desirable result that we can view *construct* as a procedure that is not tail recursive in the sense expressed in the theorem *construct-assign*. Although we leave out a few helper theorems (such as $\text{equalish}(x, \text{make-array}(j))$ is always true), the B&MTP is ready to prove theorem *equalish-construct* using induction.

THEOREM: construct-assign

$(j \notin c)$
 $\rightarrow (\text{construct}(\text{assign}(a, j, v), o, c) = \text{assign}(\text{construct}(a, o, c), j, v))$

THEOREM: equalish-construct

$\text{subbagp}(oo, \text{pair}(i, 0))$
 $\rightarrow \text{equalish}(\text{construct}(\text{make-array}(j), \text{first}(oo), \text{second}(oo)), i)$

The theorem *numnotsymbol-construct* establishes that SYMBOL is not a member of $\text{doconstruct}(\text{first}(oo), \text{second}(oo))$. Again, in the interest of space we leave out a few theorems needed to establish this fact. This leads us directly to the theorem *construct-equal*, and we are almost done proving the main theorem we wish to establish in this section.

THEOREM: numnotsymbol-construct

$(\text{inrange}(c, j)$
 $\wedge \text{unique}(c)$

$$\begin{aligned}
& \wedge (\text{SYMBOL} \notin o) \\
& \wedge (j \in \mathbf{N}) \\
& \wedge (\text{length}(c) = \text{length}(o)) \\
& \rightarrow (\text{numnotsymbol}(\text{construct}(\text{make-array}(j), o, c)) = \text{length}(c))
\end{aligned}$$

THEOREM: construct-equal

$$\begin{aligned}
& (\text{slistp}(i) \\
& \wedge \text{subbagp}(oo, \text{pair}(i, 0)) \\
& \wedge (\text{SYMBOL} \notin i) \\
& \wedge (j = \text{length}(i)) \\
& \wedge (\text{length}(oo) = \text{length}(i))) \\
& \rightarrow (\text{construct}(\text{make-array}(j), \text{first}(oo), \text{second}(oo)) = i)
\end{aligned}$$

The theorem *contents-equal* establishes that if `SYMBOL` is not an element of a , then the function call $\text{contents}(a, 0, \text{length}(a) - 1)$ returns the list a that is passed to it. The next theorems follow directly from the work that we have already done.

THEOREM: contents-equal

$$(\text{slistp}(a) \wedge (\text{SYMBOL} \notin a)) \rightarrow (\text{contents}(a, 0, \text{length}(a) - 1) = a)$$

THEOREM: pordered-first

$$\text{pordered}(oo) \rightarrow \text{ordered}(\text{first}(oo))$$

THEOREM: certsrt-completeness

$$\begin{aligned}
& (\text{permutationp}(oo, \text{pair}(i, 0)) \\
& \wedge (\text{SYMBOL} \notin i) \\
& \wedge \text{slistp}(i) \\
& \wedge \text{pordered}(oo)) \\
& \rightarrow \text{certsrt}(i, \text{first}(oo), \text{second}(oo))
\end{aligned}$$

Having proven *certsrt-completeness*, we are ready to describe the function which generates an output and a certification trail that satisfies *certsrt*. Although we do not present the program here, it is possible to define and prove correct a mergesort program that sorts a list of ordered pairs by first component. For our discussion we use a program called *pairmergesort*, which is a slight variant of a mergesort program that was proven correct by Matt Kaufmann of Computational Logic Incorporated. Given *certsrt-completeness* it is easy to see how to proceed. We define the function *first-phase* so that it takes a list i as input, and uses *pairmergesort* to sort $\text{pair}(i, 0)$. It then returns $\text{first}(oo)$ as the output, and $\text{second}(oo)$ as the certification trail, where oo is $\text{pair}(i, 0)$ sorted. The theorem *pairmergesort-works* follows immediately from the correctness of *pairmergesort* and *certsrt-completeness*.

DEFINITION:

first-phase (i)

= list (first (pairmergesort (pair (i , 0))), second (pairmergesort (pair (i , 0))))

THEOREM: pairmergesort-works

((SYMBOL $\notin i$) \wedge slistp (i))

\rightarrow certsort (i , car (first-phase (i)), cadr (first-phase (i)))

Chapter 8

Conclusion

In this thesis, we have presented a variety of programs for checking the output of other programs. Most of our work used the certification trail technique, where a program is intentionally modified to output additional information which is intended to simplify the checking process.

We presented off-line program checkers for checking sequences of mergeable priority queue operations, and also for checking sequences of splittable priority queue operations. Both ran in linear time, though some technical restrictions were placed on the sequences handled by the splittable priority queue program checker. Using the mergeable priority queue program checker as a subroutine, we showed how the problem of constructing alphabetic search trees could be efficiently certified, with the second phase checker program running in only linear time. An empirical evaluation of the mergeable priority queue program checker also showed that it ran substantially faster than the best mergeable priority queue implementations that we could find.

Using the certification trail technique, we presented on-line program checkers for mergeable priority queues and splittable priority queues. The first phase programs for both mergeable priority queues and splittable priority queues ran in $O(\log n)$ time per operation. The second phase program for mergeable priority queues ran in amortized $O(A(n))$ time per operation, where $A(n)$ is the inverse of Ackermann's function, and the second phase program for splittable priority queues ran in only $O(1)$ time per operation.

We also presented an on-line certification trail solution for performing approximate nearest neighbor queries. The first phase ran in $O(\log n)$ time per query, and the second phase ran in $O(1)$ time per query.

Next, we turned our attention to parallel computation. We showed how any parallel program can be certified so that the second phase runs in $O(1)$ time while performing the same total work as the first phase. Also, we presented a certification trail solution for the problem of evaluating a sequence of set manipulation operations. This result has applications to certifying the decision problem of determining if there is any intersection in a set of isothetic line segments, and also for certifying the finding of the maximal points in a three-dimensional point set. Using different techniques, we showed how lowest common ancestor queries, and also range maxima queries could be efficiently certified.

Finally, we discussed the possibility of combining certification trails with

traditional formal verification techniques. Specifically, we analyzed the possibility of using a software system which implements a certification trail solution for a problem, where only the second phase program has been formally verified. This results in a system which is guaranteed to never produce an incorrect output, though sometimes it may not be able to produce the correct output due to a software bug in the first phase. The savings in terms of the running time of the system, and also in the overhead needed to perform the formal verification can be substantial. To test these ideas we formally verified a certifier program for the sorting problem.

In short, this thesis has greatly increased the applicability of the certification trail technique. There are several possible directions for future research. The experimental work in this thesis could be expanded by performing an extensive empirical study of the approximate nearest neighbor certifier, the on-line mergeable priority queue certifier, and others. A formally verified certifier for the array-indexed doubly-linked list data structure would be an interesting next step for exploring the application of certification trails when providing formal verified software systems. Finally, certifications trails could be developed for more abstract data types and algorithms.

Bibliography

- [1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., "Data Structures and Algorithms", Addison-Wesley, 1983, pp. 234-239.
- [2] Alon, N., and Schieber, B., "Optimal Preprocessing for Answering On-Line Product Queries," *Technical Report, Tel Aviv University*.
- [3] Amato, N. M., Loui, M. C., "Checking Linked Data Structures," *Digest of the 24th IEEE Symposium on Fault-tolerant Computing*, 1994, pp. 164-173.
- [4] Atallah, M. J., Cole, R., and Goodrich, M. T., "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *SIAM Journal on Computing* Vol. 18, No. 3, 1989, 499-532.
- [5] Atallah, M. J., Goodrich, M. T., and S.R. Kosaraju, "Parallel Algorithms for Evaluating Sequences of Set-Manipulation Operations," *Lecture Notes 319: AWOC 88*, Springer-Verlag, 1988, pp. 1-10.
- [6] Anderson, T., and Lee, P., *Fault Tolerance: Principles and Practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [7] Avizienis, A., "The N-version Approach to Fault Tolerant Software," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.
- [8] Avizienis, A., and Kelly, J., "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, vol. 17, pp. 67-80, Aug., 1984.
- [9] Awerbuch, B., and Shiloach, Y., "New Connectivity and MSF Algorithms for Ultracomputer and PRAM", *IEEE ICPP* 1983 pp. 175-179.
- [10] Awerbuch, B., and Varghese, G., "Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pp. 258-267.
- [11] Babai, L., Fortnow, L., Levin, L. A., and Szegedy, M., "Checking Computations in Polylogarithmic Time," *Proceedings of the 23rd ACM Symposium on the Theory of Computing*, 1991, pp. 21-31.
- [12] Berkman, O., Breslauer, D., Galil, Z., Schieber, B., and Vishkin, U., "Highly Parallelizable Problems," *Proceedings of the 21st ACM Symposium on the Theory of Computing*, 1989, pp. 309-319.
- [13] Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M., "Checking the Correctness of Memories," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pp. 90-99.

- [14] Blum, M., and Kannan, S., "Designing Programs that Check their Work", *Proceedings of the 21st ACM Symposium on the Theory of Computing*, 1989, pp. 86-97.
- [15] Blum, M., Luby, M. and Rubinfeld R., "Self-Testing/Correcting with Applications to Numerical Problems," *Journal of Computer and System Sciences* **47** (1993), 549-595.
- [16] Boas, P. Van Emde, "Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space," *Information Processing Letters* **6** (1977), 80-82.
- [17] Boas, P. Van Emde, Kaas, R., and Zijlstra. E., "Design and Implementation of an Efficient Priority Queue," *Mathematical Systems Theory* **10** (1977), 99-127.
- [18] Boyer, R. S., and Moore, J. S., *A Computational Logic*, Academic Press, New York, 1979.
- [19] Boyer, R. S., and Moore, J. S., *A Computational Logic Handbook*, Academic Press, Boston, 1988.
- [20] Boyer, R. S., and Yu, Y., "Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor," Technical Report TR-91-33, Computer Science Department, University of Texas, Austin, November 1991.
- [21] Bright, J. D., "Range-restricted Mergeable Priority Queues," *Information Processing Letters* **47** (1993), 159-164.
- [22] Brown, M. R., "Implementation and Analysis of Binomial Queue Algorithms," *SIAM Journal Computing*, **7** (1978), 298-319.
- [23] Callahan, P. C., and Kosaraju, S. R., "A Decomposition of Multi-Dimensional Point-sets with Applications to k -nearest-neighbors and n -body Potential Fields," *Proceedings of the 24th ACM Symposium on the Theory of Computing*, 1992, pp. 546-556.
- [24] Chazelle, B., and Guibas, L. J., "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica* Vol. 1, No. 2, pp. 133-162.
- [25] Chen, L., and Avizienis A., "N-version Programming: a Fault Tolerant Approach to Reliability of Software Operation," *Digest of the 8th IEEE Symposium on Fault-tolerant Computing*, 1978, pp. 3-9.
- [26] Cheriton, D., and Tarjan, R. E., "Finding Minimum Spanning Trees," *SIAM Journal Computing*, **5** (1976), 724-742.

- [27] Cole, R., "Parallel Merge Sort," *SIAM Journal Computing* Vol. 17, No. 4, 1988, pp. 770-785.
- [28] Cole, R., and Vishkin, U., "Approximate Scheduling, Exact Scheduling, and Applications to Parallel Algorithms", *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 1986, pp. 478-491.
- [29] Cormen, T. H., Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1990.
- [30] Dietz, P. F., and Sleator, D. D., "Two Algorithms for Maintaining Order in a List," *Proceedings of the 19th ACM Symposium on the Theory of Computing*, 1987, pp. 365-372.
- [31] Dixon, B., Rauch, M., and Tarjan, R. E., "Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time," *SIAM Journal Computing*, **21** (1992), 1184-1192.
- [32] Dixon, B., *Personal Communication*, July 1993.
- [33] Edmonds, J., "Matroid Partition," *Lectures in Applied Mathematics - Mathematics of the Decision Sciences*, George B. Dantzig and Arthur F. Veinott, Jr., EDITORS **11** (1968), pp. 335-345
- [34] Fortnow, L., "The Complexity of Perfect Zero Knowledge," *Proceedings of the 19th ACM Symposium on the Theory of Computing*, 1987, pp. 204-209.
- [35] Fredman, M. L., *et al.*, "The Pairing Heap: a New Form of Self-Adjusting Heap," *Algorithmica*, **1** (1986), 111-129.
- [36] Fredman, M. L., and Tarjan, R. E., "Fibonacci Heaps and Their Uses in Network Optimization," *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, 1984, pp. 338-345.
- [37] Fredman, M. L., and Willard, D. E., "Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths", *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, 1990, pp. 719-725.
- [38] Gabow, H. N., Bentley, J. L., and Tarjan, R. E., "Scaling and Related Techniques for Geometry Problems," *Proceedings of the 16th ACM Symposium on the Theory of Computing*, 1984, pp. 135-143.
- [39] Gabow, H. N., Galil, Z., Spencer, T. H., and Tarjan, R. E., "Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs", *Combinatorica* **6** (1986), pp. 109-122.

- [40] Gabow, H. N., and Tarjan, R. E., "A Linear-Time Algorithm for a Special Case of Disjoint Set Union," *Journal Computer System Sciences*, **30** (1985), 209-221.
- [41] Gemmell, P., Lipton, R., Rubinfeld, R., Sudan, M., and Wigderson, A., "Self-testing/correcting for Polynomials and for Approximate Functions," *Proceedings of the 23rd ACM Symposium on the Theory of Computing*, 1991, pp. 32-42.
- [42] Goldwasser, S., Micali, S., and Rackoff, C., "The Knowledge Complexity of Interactive Proofs," *SIAM Journal Computing*, Vol. 18, 1989, 186-208.
- [43] Graham, R. L., and Hell, P., "On the History of the Minimum Spanning Tree Problem," *Ann. Hist. Comput.*, pp. 43-47, Jan., 1985.
- [44] Guibas, L., Hershberger, J., Leven, D., Sharir, M., Tarjan, R. E., "Linear Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons," *Algorithmica* **2** (1987), 209-223.
- [45] Guzmán, J., and Hudak, P., "Single-threaded Polymorphic Lambda Calculus," *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1990.
- [46] Hoare, C. A. R., "Quicksort," *Computer Journal*, pp. 10-15, 5(1), 1962.
- [47] Honsell, F., Mason, I. A., Smith, S., and Talcott, C., "A Variable Typed Logic of Effects," *Information and Computation, to appear*.
- [48] Hu, T. C., *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.
- [49] Huang, K.-H., and Abraham, J., "Algorithm-based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, pp. 518-529, vol. C-33, June, 1984.
- [50] JáJá, J., *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.
- [51] Johnson, B. W., *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, Reading, MA, 1989.
- [52] Jones, D. W., "An Empirical Comparison of Priority Queue and Event Set Implementations," *Communications of the ACM*, **29** (1986), 300-311.
- [53] Kaufmann, M., "Response to FM91 Survey of Formal Methods: Nqthm and Pc-Nqthm," Technical Report 75, March 1992, Computational Logic Inc.
- [54] Klein, P. N., and Tarjan, R. E., "A Randomized Linear-Time Algorithm for Finding Minimum Spanning Trees," *Proceedings of the 26th ACM Symposium on the Theory of Computing*, 1994, pp. 9-15.

- [55] Kosaraju, S. R., and Delcher, A. L., "Optimal Parallel Evaluation of Tree-Structured Computations by Raking," *Lecture Notes in Computer Science*, 319 pp. 101-110.
- [56] Knuth, D. E., *The Art of Computer Programming*, Vol. 1, 2nd edn., Addison-Wesley, Reading, MA, 1973.
- [57] Knuth, D. E., *The Art of Computer Programming*, Vol. 3, 2nd edn., Addison-Wesley, Reading, MA, 1973.
- [58] Kruskal, C. P., Rudolph, L., and Snir, M., "Efficient Parallel Algorithms for Graph Problems", *Algorithmica*, (1990) 5: pp. 43-64.
- [59] Kruskal, J. B., "On the Shortest Spanning SubTree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, 7 (1956), 48-50.
- [60] Ramanan, P., "Testing the Optimality of Alphabetic Trees," *Theoretical Computer Science*, (1992) 93: pp. 279-301.
- [61] Kwan, S., and Ruzzo, W. L., "Adaptive Parallel Algorithms for Finding Minimum Spanning Trees", *Proceedings of the 1984 International Conference on Parallel Processing*, Aug. 1984, pp. 439-443.
- [62] Liao, A. M., "Three Priority Queue Applications Revisited," *Algorithmica*, 7 (1992), 415-427.
- [63] Mahmood, A., McCluskey, E., "Concurrent Error Detection Using Watchdog Processors - a Survey," *IEEE Transactions on Computers*, pp. 160-174, vol. 37, February, 1988.
- [64] Nair, V. S. S., and Abraham, J. A., "General Linear Codes for Fault-tolerant Matrix Operations on Processor Arrays," *Digest of the 18th IEEE Symposium on Fault-tolerant Computing*, 1988, pp. 180-185.
- [65] Preparata, F. P., and Shamos, M. I., "Computational Geometry", Springer-Verlag, 1985.
- [66] Prim, R. C., "Shortest Connection Networks and Some Generalizations," *Bell Systems Technical Journal*, pp. 1389-1401, Nov., 1957.
- [67] Purushothaman, S., and Subrahmanyam, P. A., "Mechanical Certification of Systolic Algorithms," *Journal of Automated Reasoning*, 5 (1989), 67-91.
- [68] Randell, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

- [69] Rubinfeld, R. A., "A Mathematical Theory of Self-checking, Self-testing and Self-correcting Programs," Ph.D. Dissertation, University of California, Berkeley, 1990.
- [70] Sastry, A. V. S., Clinger, W., and Ariola, Z., "Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates," *Conference on Functional Programming Languages and Computers*, pp. 266-275, 1993.
- [71] Schieber, B., and Vishkin, U., "On Finding Lowest Common Ancestors: Simplification and Parallelization," *SIAM Journal Computing*, vol. 17, 1988, 1253-1262.
- [72] Sedgewick, R., "Implementing Quicksort Programs," *Communications of the ACM*, pp. 847-857, 21(10), 1978.
- [73] Sleator, D. D., and Tarjan, R. E., "Self-Adjusting Heaps," *SIAM Journal on Computing*, **15** (1986), 52-69.
- [74] Sudan, M., and Wigderson, A., "Self-Testing/Correcting for Polynomials and for Approximate Functions," *Proceedings of the 23rd ACM Symposium on the Theory of Computing*, 1991, pp. 32-42.
- [75] Sullivan, G. F., and Masson, G. M., "Using Certification Trails to Achieve Software Fault Tolerance," *Digest of the 20th IEEE Symposium on Fault-tolerant Computing*, 1990, pp. 423-431.
- [76] Sullivan, G. F., and Masson, G. M., "Certification Trails for Data Structures," *Digest of the 21st IEEE Symposium on Fault-tolerant Computing*, 1991, pp. 240-247.
- [77] Sullivan, G. F., Wilson, D. S., and Masson, G. M., "Certification Trails and Software Design for Testability," *Proceedings of the 1993 International Test Conference*, pp. 200-209.
- [78] Tarjan, R. E., "Efficiency of a Good But Not Linear Set Union Algorithm," *Journal of the ACM*, Vol. 22, No. 2, 1975, pp. 215-225.
- [79] Tarjan, R. E., "Applications of Path Compression on Balanced Trees," *Journal of the ACM*, Vol. 26, No. 4, October 1979, pp. 690-715.
- [80] Tarjan, R. E., Vishkin, U., "Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time", *SIAM Journal Computing*, vol. 14, 1985, 862-874.

- [81] Taylor, D., "Error Models for Robust Data Structures," *Dig. 20th IEEE Symposium on Fault-tolerant Computing*, 1990, pp. 416-422.
- [82] Vuillemin, J., "A Data Structure for Manipulating Priority Queues," *Communications of the ACM*, **21** (1978), 309-314.
- [83] Vuillemin, J., "A Unified Look at Data Structures," *Communications of the ACM* 1980, pp. 229-239.
- [84] Wilson, D. S., Sullivan, G. F., and Masson, G. M., "Experimental Evaluation of Certification Trails Using Abstract-data-type Validation," *Proceedings of the IEEE Computer Software and Applications Conference*, 1992, pp. 300-306.
- [85] Yao, A. C. C., "Coherent Functions and Program Checkers," *Proceedings of the 22nd ACM Symposium on the Theory of Computing*, 1990, pp. 84-94.

Vita

Jonathan David Bright was born in Brooklyn, New York, on August 24th, 1967. He studied Computer Science at the University of Delaware, where he earned a B.S. in 1988. After graduation, he went directly to the Johns Hopkins University to begin graduate work. After playing much chess and bridge, and learning to Swing Dance, he completed the requirements for his Ph.D. in Computer Science in October, 1994. Jon went to Bell Labs in Murray Hill after graduation.