On-line Error Monitoring for Several Data Structures

Jonathan D. Bright[1]

AT&T Bell Laboratories, Murray Hill, New Jersey 07974

Gregory F. Sullivan[2]

Computer Science Department, Johns Hopkins University Baltimore, Maryland 21218

We present several examples of programs which efficiently monitor the answers from queries performed on data structures to determine if any errors are present. Our paper includes the first efficient on-line error monitor for a data structure designed to perform nearest neighbor queries. Applications of nearest neighbor queries are extensive and include learning, categorization, speech processing, and data compression. Our paper also discusses on-line error monitors for priority queues and splittable priority queues. On-line error monitors immediately detect if an error is present in the answer to a query. An error monitor which is not on-line may delay the time of detection until a later query is being processed which may allow the error to propagate or may cause irreversible state changes. On-line monitors can allow a more rapid and accurate response to an error.

1 Introduction

A richly diverse and still growing collection of techniques have been developed to perform software based fault tolerance including: recovery blocks [18], N-version programming [3], program checkers [4, 5], algorithm-based methods [12, 16], robust data structures [13, 14, 24, 25, 26], certification trails [20, 21, 22, 23, 27], and other methods [1]. In this paper, we consider the problem of detecting errors in the answers given in response to data structure queries. For many programs a substantial fraction of the intricate error-prone code resides in the routines associated with data structure access. In addition, a significant fraction of total program running time is spent executing the code of these routines. Data structures are often used to implement abstract data types (ADTs) and in recent years a number of different ADTs have been shown to be efficiently checkable. Checkers which are also referred to as error monitors in this paper have been developed for priority queues, the union/find data type, and most recently mergeable priority queues and have been shown to require only $O(n)$ time to check a sequence of n operations [5, 21, 23]. These checkers detect any errors in query answers and since each of these checkers is based on an ADT definition it works for any data structure implementation of the ADT.

This generality is useful and powerful but in some cases it may have a cost in terms of efficiency and/or latency. The ADT checkers developed to date often operate in an off-line fashion. That is, a sequence of n operations together with all of the supposed answers are given to the checker in one batch. The checker then processes all of the operations and the answers to determine if all of the supposed answers are correct. While useful processing can be done as each operation is received, it is still the case that for the three ADT checkers mentioned above errors are not guaranteed to be caught until after the last operation is received and a final processing step is completed.

One response to this concern is to envision supplying the checker with carefully chosen supplementary data in addition to the answers given by the ADT with the hope that the checker can exploit this added information to perform its task more rapidly. This basic idea was one of the motivations for the introduction of the *certification trail* technique [20, 23]. Here, a program which is called the first phase computes the answer to the problem, but is also modified to output additional information called a certification trail. A second phase program reads the input, the answer, and the certification trail. Using the certification trail, the second phase determines if the answer is correct or incorrect. If the inputs to the first phase are all correct then the checker accepts the answer as correct. Note, however, an important property of the certification trail technique is that the second phase will never accept incorrect answers, even if the certification trail it is given is also incorrect.

We will describe some certification trail solutions which allow error monitoring to be performed on-line. For such schemes to work effectively it is important that the cost of providing the additional data, i.e., the certification trail, is not excessive. However, the cost of providing this data may depend on the type of data structure used to implement the ADT. For this reason, in this paper we speak of error monitoring for data structures instead of error monitoring for ADTs.

For ease of exposition, we begin by illustrating our ideas on a structure which consists of multiple distinct instances of a basic search data structure. Each instance allows operations such as *insert*, *delete*, *predecessor*, *successor* and *min*. Note, this includes the functionality of a priority queue. We show how operations can be certified so that the checker phase detects errors in an on-line fashion, and can run in only $O(1)$ time per operation. This is a slight generalization of the result for a single instance of the search data structure which was implicit in an earlier paper [20]. Next we build on this result to design an error monitor for a splittable search data structure (SSDS). In addition to the operations above an SSDS supports the operation *split*. Note, this includes the functionality of a spittable priority queue. The second phase checker program we construct runs in only $O(1)$ time per operation.

[1] This research was supported by the 1993 CESDIS Cray Research Earth and Space Science Fellowship.

[2] This research was supported by NSF grant CCR-9319945.

The final data structure we discuss is called a multidimensional search tree. Multidimensional search trees have a variety of uses, and in this paper we discuss checking the correctness of nearest neighbor queries, and a recently introduced variant called approximate nearest neighbor queries [6, 8, 2]. Applications of nearest neighbor queries are extensive and include learning, categorization, speech processing, and data compression [8, 11]. For the case of approximate nearest neighbor queries, the second phase checker program runs in only $O(1)$ time per operation.

There are a variety of techniques in the fault tolerance area which are related to program checkers and certification trails and we will now briefly discuss three. The first technique is called algorithm-based fault tolerance [12][16]. 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.

The second idea that we discuss is that of an acceptance test [18]. 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. Acceptance tests are part of a larger scheme called recovery blocks [18]. One may structure a program so that it consists of a set of recovery blocks which may be disjoint or nested. An acceptance test can be placed at the end of a recovery block so that if an error is discovered an alternate method for computing the answer can be attempted. The programs discussed in this paper can function as acceptance tests.

Finally, we mention the notion of a repairable or robust data structure [13, 14, 24, 25, 26]. In this technique, a data structure is augmented with additional information, such as redundant values and links. The idea is that if the data structure should become corrupted, then by using the redundant information it may be possible to restore the data structure to its correct form. Some of the proposed robust data structures include error detection mechanisms. One could use programs such as those discussed in this paper to perform error monitoring for robust data structures which do not have error monitors or which have inefficient error monitors. The programs could also be used as supplementary error detectors. When an error is detected a repair of the data structure could be automatically undertaken.

1.1 The Certification Trail Technique

A formal description of the certification trail technique is given below. Note, this is a slight reformulation of the original certification trail definition [20] 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. Some fault tolerance techniques, such as watchdog monitor processors[15], are very effective at detecting program failures, but can be ineffective at detecting failures which lead only to incorrect output. The advantage of certification trails over techniques such as these is that faults which lead only to incorrect answers are guaranteed to be detected, provided the faults are isolated to one execution.

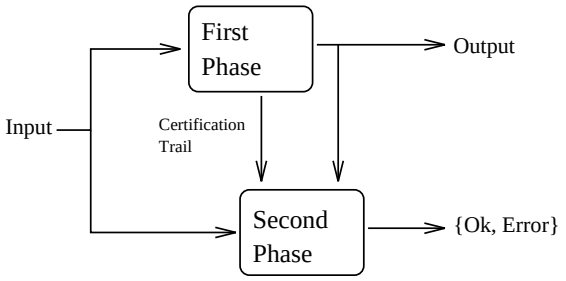


Figure 1: Illustration for the certification trail definition.

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:

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. However, the definition allows the second phase to accept a correct output even when the certification trail is not of the correct form. In the following discussions, we refer to the program F_1 as the first phase and F_2 as the second phase. See Figure 1 for an illustration.

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 [20, 21, 22, 23, 27]. 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%.

2 Basic Definitions and Search Data Structures

The programs presented in this paper perform checks "on-line". By "on-line" we mean that after every operation the second phase must output either *ok* or *error* before the next operation is processed. In order to construct efficient checkers we found it necessary to use the *certification trail* method of [20][21]. That is, we modify the first phase to also output some additional information called a certification trail to aid the second phase. When referring to the time complexity of first and second phases we use the notation $(f(n), g(n))$. This is shorthand for saying that the first phase runs in $O(f(n))$ time and the second phase runs in $O(g(n))$ time. We use this notation to express both amortized and worst case complexity of certification trail algorithms.

The basic operand type for search data structures is a pair (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 is 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 that 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, throughout this paper we refer to *pairs* as *elements*. Sets of pairs are referred to by a set name, which is assumed to be a positive integer. We now describe the operations that we consider.

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

2. $insert((it, val), S)$: Insert the pair (it, val) into 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 the set it is contained in. A 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. $pred((it, val), S)$: Return the pair in set S with the largest value that is less than (it, val). If none of the pairs in S have value less than (it, val) then return the special symbol *smallest*.

6. $succ((it, val), S)$: Return the pair in set S with the smallest value that is greater than (it, val). If none of the pairs in S have value greater than (it, val) then return the special symbol *largest*.

7. $deletemin(S)$: 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.

8. $split(S, val, S_1, S_2)$: Form the set S_1 be 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 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.

Sometimes we will use longer more descriptive names for operations, e.g., *predecessor* for *pred* and *successor* for *succ*. All of the operations which do not return pairs are assumed to return the value *nil*. The requirement that set names and item numbers not be reused is for the ease of describing our algorithms. In all cases it is possible to modify our algorithms so that item numbers and set names can be reused. We say that a set S is *active* if it has been created by a *makeset* or *split* operation, but has not yet been destroyed.

A data structure which supports the operations *makeset*, *insert*, *delete*, *min*, *pred*, *succ*, and *deletemin* we call a *search data structure*. Finally, a data structure that supports search operations, and in addition supports the *split* operation we call a *splittable search data structure*.

3 Certifying Non-splittable Search Data Structures

Before describing how to certify splittable search data structures we discuss the simplified problem of certifying search data structures. As defined above a *search data structure* consists of multiple instances of a basic searchable data structure with *split* operations disallowed. The certification trail solution consists of two phases which we will now describe. 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. See Figure 2 for an example. 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 maintained in sorted order 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*. Figure 2 illustrates the insertion of a pair.

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

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 four checks: (i) $where[it']$ is not nil, (ii) the node pointed to by $where[it']$ stores the pair (it', val') and also has a set name field of S, (iii) $(it', val') < (it, val)$ holds, (iv) the pair in the node after the node containing (it', val') in the doubly linked list is greater than or equal to (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

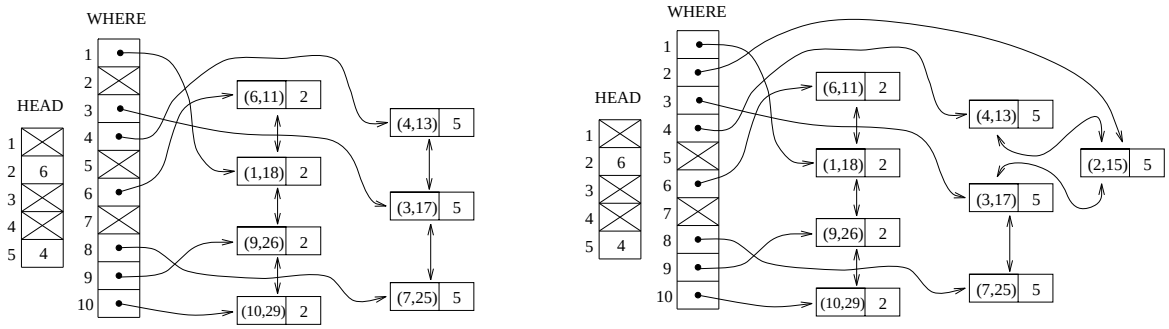
Now consider the operation $insert((it, val), S)$. Given no additional information, there is no efficient method for the second phase to determine the correct location in the doubly linked list for S in which to insert (it, val). Thus, the second phase reads the certification trail for this problem. 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 $where[it']$ indicates the node after which the new pair is to be inserted. 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) including the set name 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.

It is not difficult to show that the procedures described above allow one to accurately check the correctness of a sequence of operations and queries on a search data structure. The first phase is a straightforward implementation of the search data structure ADT which also outputs additional information, i.e., the certification trail. The second phase properly uses the information from the certification trail. It is not hard to prove the invariant that the doubly linked list associated with each set is always maintained in sorted order and it contains precisely the pairs currently in the set. One can show that incorrect information on the certification trail will not cause a pair to be inserted improperly because of the checks that are performed.

Lemma 3.1 *Search data structures (without split operations) 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.*

We omit the proof here because of space limitations and because a similar proof appears in the next section. The running times given for the first phase and the second phase follow directly from the implementations described.



List 2 contains the elements (6,11) (1,18) (9,26) (10,29).

List 5 contains the elements (4,13) (3,17) (7,25).

After the element (2,15) is inserted into list 5.

Figure 2: Example of an array indexed set of doubly linked lists.

4 Certifying Splittable Search Data Structures

In this section we consider checking a sequence of operations on a splittable search data structure in an on-line fashion. At first glance it might appear that an approach identical to that used in the previous section will work. But there is a subtlety associated with the *split* operation. Let us consider the time complexity of the second phase described above. In the second phase each node in a doubly linked list has a field which records the set name in which a pair is contained. The cost of updating these fields when a *split* operation occurs would be linear in the number of pairs in the set being split since each node would have to record one of the two new set names. In the worst case the complexity of processing n operations would become quadratic instead of linear.

One way to try and avoid this time penalty is to remove the field which contains the set name from the nodes in the doubly linked lists. If the fields are omitted then they clearly will not require updating after a *split* operation. But this approach causes another problem. Note, whenever a pair is inserted into a set during the second phase checks are performed to make sure that a node adjacent to the inserted element in the doubly linked list has the proper set name recorded. If the set name field is omitted then this check can not be performed. Without this check an incorrect certification trail could result in a pair being inserted into a doubly linked list associated with the wrong set. This could then cause the second phase to output ok even when an answer was incorrect, i.e., an incorrect certification trail could 'fool' the second phase which is unacceptable.

To avoid this problem we must reformulate the second phase. All the doubly linked lists are combined into a single doubly linked list with special 'marker' nodes. The set name fields are omitted from the nodes. Below we show how to use the powerful list ordering result of Dietz and Sleator [9] to perform tests which prevent the incorrect insertion problem described above. The two phases of the full solution are described below by 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 [7]. 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 are used to delimit the elements of each set. Two arrays called *start* and *end* which are indexed by set names are used to point to these markers. So $start(S)$ and $end(S)$ appear immediately before and after the elements in L corresponding to the set S. These markers will be used to help test in which set a pair is contained. Finally, we maintain the array *where* indexed by item numbers so that given an item number *it*, $where[it]$ contains a pointer to the node in L which stores the pair with item number *it*. See Figure 3 for an example. For each operation, let *ans* be the supposed answer and *cert* the certification trail, and perform the appropriate step below.

$makeset(S)$:
Insert the markers $start(S)$ and $end(S)$

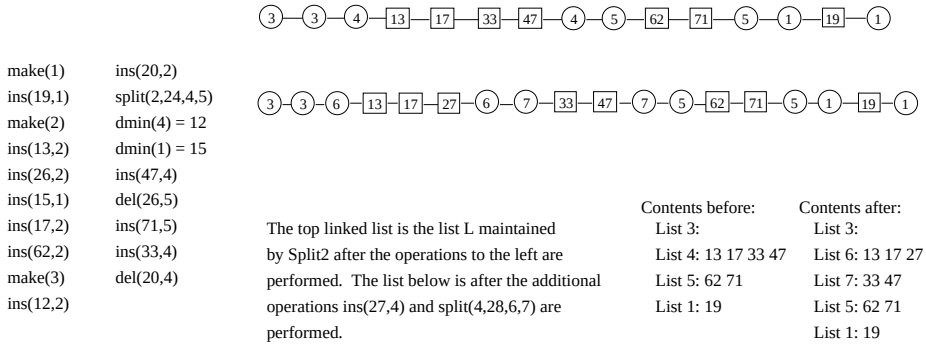


Figure 3: Representation of the splittable search data structure for the second phase. To enhance readability item numbers are suppressed and just values appear. Arrays *where*, *start*, and *end* are not shown.

at the beginning of L.

insert$((it, val), S)$:
IF ($cert = nil$) THEN
Check that if S is not empty then 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 the pointer $where[it']$ falls between the pointers $start(S)$ and $end(S)$ in L.
Check that the node pointed to by $where[it']$ contains (it', val').
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 the pointer $where[it']$ falls between the pointers *start(S)* and *end(S)* in L.
Check that the node pointed to by $where[it']$ contains (it', val').
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 the pointer $where[it']$ falls between the pointers *start(S)* and *end(S)* in L.
Check that the node pointed to by $where[it']$ contains (it', val').
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 the pointer $where[it']$ falls between the pointers $start(S)$ and $end(S)$ in L.
Check that the node pointed to by $where[it']$ contains (it', val').
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.

The operation *Deletemin* can simulated by performing a *min* followed by a *delete*.

We now discuss the implementation of $CertSSDS_1$ and $CertSSDS_2$. The only difficulty occurs in $CertSSDS_2$ when we need to perform checks such as determining whether the pointer $where[it']$ falls between the pointers $start(S)$ and $end(S)$ in L. For this we use a result of Dietz and Sleator [9] 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. Their most powerful result achieves constant time for both update and query operations on a list. The authors also present much a simpler algorithm which achieves constant running time per operation in an *amortized*[7] sense. The reader the should consult [9] for details on these algorithms, and also heuristics that will likely work well in practice. 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 4.1 *$CertSSDS_1$ 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.*

The proof of this lemma will appear in the final version of the paper.

5 Certifying Nearest Neighbor Queries

In this section 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 $dist(q, p) \leq dist(q, p_{opt}) * (1 + \epsilon)$, where p_{opt} is the actual nearest neighbor of q in the set. See Figure 4 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 [17]. 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

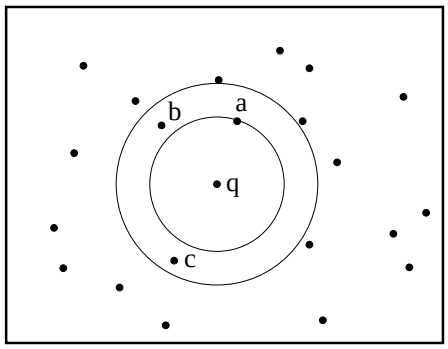


Figure 4: Example of approximate nearest neighbor query.

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 tree. When a particular splitting strategy produces a highly unbalanced tree, it may be necessary to artificially balance the tree using a centroid decomposition [10] 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 [6], 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.

5.1 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 $P = \{p_1, p_2, ..., p_k\}$ of the set of points such that for all p_i in P, $dist(q, p_i) \leq dist(q, p^k_{opt}) * (1 + \epsilon)$, where p^k_{opt} is the actual kth 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 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.

5.2 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 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 in P_A.

The fifth condition, together with the definition of the sphere S_i, implies that the kth 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 kth 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 [19]). 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.
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 5 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 rectangle 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

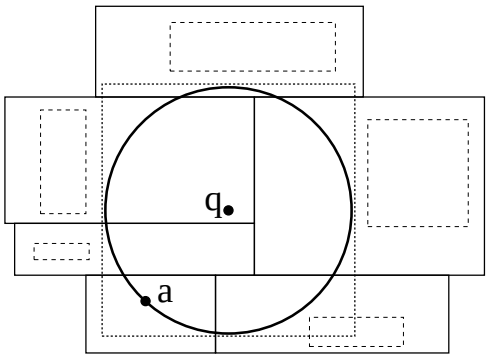


Figure 5: Example of covering the box around the sphere.

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 5.1 *Let $r_1, r_2, ..., r_n$ be a set of d-dimensional rectangles, with each rectangle r_i represented as a Cartesian product $[r_i(x_1), r_i(x'_1)] \times [r_i(x_2), r_i(x'_2)] \times \ldots \times [r_i(x_d), r_i(x'_d)]$ in $\Re^d$. 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 jth dimension. Let B be the set of the coordinates of all of the rectangles in the jth dimension. (So, for each r_i, both $r_i(x_j)$ and $r_i(x'_j)$ appear in B.) Let $b_1 < b_2 < \ldots < b_m$ 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 $r'_1, r'_2, ..., r'_n$ from $r_1, r_2, ..., r_n$ as follows: r'_i has the same coordinates as r_i except in the jth dimension, where a coordinate with value b_k is replaced with value l.*

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

A proof of this lemma appears in the final version of the paper.

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 section in the following lemma.

Lemma 5.2 *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.

6 Conclusion

In this paper, we developed certification trail solutions for some significant and versatile data structures. For our first two results, we presented certifiers for search data structures and splittable search data structures which ran in only $O(1)$ time per operation. Our next result was for multidimensional search trees, and we showed how approximate nearest neighbor queries could be certified in only $O(1)$ time per query. Our techniques are directly applicable to exact nearest neighbor queries as well. All three certifiers check the answers given in an on-line fashion. That is, as soon as a query and an answer are received, the certifier can determine if the answer really is correct, before any more operations in the sequence are received. The advantage of having an on-line certifier is that the sooner an error is detected, the less chance there is of the error propagating to other parts of the system. We are currently trying to develop on-line certifiers for other interesting data structures.

7 References

[1] Amato, N. M., Loui, M. C., "Checking Linked Data Structures," *Dig. 24th FTCS*, 1994, pp. 164-173.

[2] Arya, S., Mount D. M., Netanyahu, N. S., Silverman, R., and Wu, A., "An Optimal Algorithm for Approximate Nearest-Neighbor Searching", *Proc. 5th Ann. Symp. Disc. Alg.*, 1994, pp. 573-582.

[3] Avizienis, A., "The N-version approach to fault tolerant software," *Trans. Soft. Eng.*, v. 11, pp. 1491-1501, 1985.

[4] Blum, M., and Kannan, S., "Designing Programs that Check their Work", *Proc. 21st STOC*, 1989, pp. 86-97.

[5] Bright, J., and Sullivan, G., "Checking Mergeable Priority Queues," *Dig. 24th FTCS*, pp. 144-153.

[6] 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," *Proc. 24th ACM STOC*, 1992, pp. 546-556.

[7] Cormen, T. H., Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms,* MIT Press, 1990.

[8] Dasarathy, B. V., "Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques", IEEE Computer Society Press, 1991.

[9] Dietz, P. F., and Sleator, D. D., "Two Algorithms for Maintaining Order in a List," *Proc. 19th ACM STOC*, 1987, pp. 365-372.

[10] 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.

[11] Gersho, A., and Gray, R. M., "Vector Quantization and Signal Compression", Kluwer Academic Publishers, Boston 1993.

[12] Huang, K.-H., and Abraham, J., "Algorithm-based Fault Tolerance for Matrix Operations," *IEEE TOC*, pp. 518-529, vol. C-33, June, 1984.

[13] Kanawati, N. A., Kanawati, G. A., Abraham, J. A., "A Modular Robust Binary Tree," DCCA-4, January 1994.

[14] Kant, K., and Ravichandran, A., "Synthesizing Robust Data Structures - An Introduction," *IEEE TOC*, pp. 161-173, vol. 39, Feb., 1990.

[15] Mahmood, A., McCluskey, E., "Concurrent Error Detection Using Watchdog Processors - a Survey," *IEEE TOC*, pp. 160-174, vol. 37, Feb., 1988.

[16] Nair, V. S. S., and Abraham, J. A., "General Linear Codes for Fault-tolerant Matrix Operations on Processor Arrays," *Dig. 18th FTCS*, 1988, pp. 180-185.

[17] Preparata, F. P., and Shamos, M. I., "Computational Geometry", Springer-Verlag, 1985.

[18] Randell, B., "System Structure for Software Fault Tolerance," *IEEE Trans. on Soft. Eng.*, vol. 1, pp. 220-232, June, 1975.

[19] Schieber, B., and Vishkin, U., "On Finding Lowest Common Ancestors: Simplification and Parallelization," *SIAM J. Comp.*, vol. 17, 1988, 1253-1262.

[20] Sullivan, G. F., and Masson, G. M., "Using Certification Trails to Achieve Software Fault Tolerance," *Dig. 20th FTCS*, 1990, pp. 423-431.

[21] Sullivan, G. F., and Masson, G. M., "Certification Trails for Data Structures," *1991 FTCS*, 240-247.

[22] Sullivan, G. F., Wilson, D. S., and Masson, G. M., "Certification Trails and Software Design for Testability," *Proc. 1993 Inter. Test Conf.*, pp. 200-209.

[23] Sullivan, G. F., Wilson, D. S., and Masson, G. M., "Certification of Computational Results," *IEEE TOC*, to appear.

[24] Taylor, D. J., Morgan, D. E., Black, J. P., "Redundancy in Data Structures: Improving Software Fault Tolerance," *Trans. Soft. Eng.*, 585-594, v. 6, 1980.

[25] Taylor, D. J., Morgan, D. E., Black, J. P., "Redundancy in Data Structures: Some Theoretical Results," *Trans. Soft. Eng.*, 595-602, v. 6, 1980.

[26] Taylor, D., "Error Models for Robust Data Structures," *Dig. 20th FTCS*, 1990, pp. 416-422.

[27] Wilson, D. S., Sullivan, G. F., and Masson, G. M., "Experimental Evaluation of Certification Trails Using Abstract-data-type Validation," *Proc. IEEE Comp. Soft. App. Conf.*, 1992, pp. 300-306.