Checking Value-Sensitive Data Structures in Sublinear Space

Michael T. Goodrich[1] and Jonathan Z. Sun[2]

[1] Department of Computer Science, University of California, Irvine, CA 92697-3435
goodrich@ics.uci.edu
[2] School of Computing, The University of Southern Mississippi
118 College Drive, Box 5106, Hattiesburg, MS 39406
jonathan.sun@usm.edu

Abstract. Checking value-sensitive data structures in sublinear space has been an open problem for over a decade. In this paper, we suggest a novel approach to solving it. We show that, in combination with other techniques, a previous method for checking value-insensitive data structures in log space can be extended for checking the more complicated value-sensitive data structures, using log space as well. We present the theoretical model of checking data structures and discuss the types of invasions a checker might bring to the data structure server. We also provide our idea of designing sublinear space checkers for value-sensitive data structures and give a concrete example – a log space checker for the search data structures (SDS).

1 Introduction

Checking the correctness of the computing results of a program with less efforts than recomputing those results finds applications in different areas of computer science. These applications include hardware and software reliability, fault tolerant computing, soundness of algorithms, data authentication, and online transaction auditing, just to name a few. Here we call a program that checks the results of another program and reports errors the *checker*, and the program under check the *checkee*. A checker is evaluated with its validity and efficiency. That is, a checker should be able to catch all the errors and pass all the correct results, and this should be done as efficiently as possible.

A checker is *time efficient* if its time complexity is lower than that of the checkee and the space complexity is not higher than that of the checkee. Similarly, we can define a checker of being *space efficient* when its space complexity is lower and time complexity is not higher. We say that a checker is *optimal* if it has space complexity logarithm of the checkee's space complexity and time complexity $O(1)$ of checking each operation performed by the checkee. (Recall that $O(\log n)$ is the information theoretic lower bound to encode n bits of information.) For example, an optimal search data structure uses $O(n)$ (linear) space and $O(\log n)$ time to do a search, insert, or delete operation.

1.1   Theoretic Model

Here we briefly introduce the theoretic model of checking data structures. Most definitions and terminologies follow [1,3,9], but the idea of classifying invasions into two types is new.

The encapsulation model. In this model, users access a data structure D through a set of operations provided by D, for example, $insert(x)$, $delete(x)$, or $search(x)$ if D is a search data structure. The data structure D performs some computation to realize the operation. For some operation, D returns a result to user, like for $search(x)$ it returns a boolean value indicating whether x is contained in D or not. D encapsulates the realization of the operations so that it is irrelevant to the users how and how efficient an operation is fulfilled in D. The checker C is in between of users and D, the checkee, audits the operations issued by users and the results returned by checkee, and reports "error" if any result of an operation is incorrect. This model assumes that D is run on untrusted media with possibility of mistakes or malicious malfunctions, however C is trustworthy. Therefore if the soundness of C is proved, then C and the unreliable D together would encapsulate a reliable D to users. A checker C in this model needs to be individually designed for each data structure D.

Invasions. Checker C is *invasive* if it requires some augmentation of D to facilitate the checking, or if it issues extra operations to D that are not issued by users. The checkers discussed in this paper and in related work are all invasive checkers. (See Section 1.2). We categorize the invasions of those checkers into the following two types:

 – *Storage Invasions.* With such invasions, C requests D to associate additional information to each data element stored in D, where the additional information is not necessary for D to perform any operation it provides. As an example, the RAM (random access memory) checker by Blum *et al.* in [3] requests each value x to be stored as a pair (x, i), with i being a discrete index indicating the time (order) of insertion.
 – *Operation Invasions.* With such invasions, C issues extra operations to D to follow up a operation issued by a user. Using again the example of Blum *et al.*'s RAM checker [3], the checker C turns each $read(x)$ into a $read(x, i)$ followed by a $write(x, i')$. In another example, the linear space SDS (search data structure) checker by Bright and Sullivan in [5] turns an $insert(x)$ into an $insert(x, i)$ followed by a $predecessor(x)$.

Excessiveness. An invasion is excessive if it increases the asymptotic space complexity of the checkee data structure D to store a data element or the asymptotic time complexity of D to fulfill an original user operation. Otherwise the invasion is non-excessive. A checker with only non-excessive invasions is a non-excessive checker. For example, following up an $insert(x)$ by an operation of $O(n)$ time is an excessive operation invasion to an efficient SDS, which ought to be able to do $insert(x)$ in $O(\log n)$ time. We are only interested in non-excessive checkers.

As two extreme cases, the following invasions are always non-excessive, regardless of the particular time and space complexities of the original unchecked data structure D.

- An storage invasion that requires only $O(1)$ size additional information to be associated to each data element.
- An operation invasion such that the extra operations following each original user operation can be realized in $O(1)$ time by D.

We call such invasions *minimal storage invasions* and *minimal operation invasions*. The checker we provide in this paper commits only minimal (storage and operation) invasions.

Value-insensitive and value-sensitive data structures. Some fundamental data structures have the property that the value stored at each data element plays no role in determining how the data is stored and queried. We call them *value-insensitive data structures*. Maps, arrays, stacks, queues, linked lists, and linked directed graphs are all value-insensitive data structures. [1] When these data structures are queried, the answer is determined solely by the sequence of operations or a specific argument in an operation, such as a memory address, an array index, or a link (pointer). In the opposite, in many advanced data structures such as heaps or binary search trees, the structure to organize data elements and the results of operations depend on a key value contained in each data element. We call these data structures *value-sensitive data structures*. (See [3] for more details.)

Blum et al.'s open problem. We conclude this brief introduction to theoretical model with the following open problem raised by Blum *et al.* in [3]. This paper will suggest and illustrate a novel approach of solving this problem.

Problem 1. Is there any checker under the encapsulation model that checks a value-sensitive data structure such as a binary search tree or heap in sub-linear space? The checker can have only non-excessive storage and operation invasions.

1.2 Related Work

Among the rich literatures on program checking, only a few papers addressed the problem of sublinear space checking. In their fundamental paper [3], Blum *et al.* gave a method of checking unreliable memory (RAM) of size n with a small reliable memory of size $O(\log(n))$, by using ϵ-biased hash functions. In the same paper, authors also applied their method to check two other value-insensitive data structures *stacks* and *queues*. Amato and Loui [1] further extended the result to work on *linked data structures* including lists, trees and general graphs.

---

[1] Despite its physical meaning, the random access memory (RAM) studied in [3] can be viewed as a map data structure. It maps a memory address to the value stored at this address and supports two operations – writing a value to an address and reading the value from an address.

These checkers all use the ϵ-biased hash functions discovered by J. Naor and M. Naor [14]. Therefore we call them *hash-based* checkers. These checkers commit storage and operation invasions but are extremely efficient. They use $O(\log n)$ space and check each operation in $O(1)$ time w.h.p. However, as pointed out by Blum *et al.* in the open problem, extending this method onto checking value-sensitive data structures is nontrivial.

Independent of Blum *et al.*'s work, some linear space checkers of value-sensitive data structures have been developed using a different technique, which we call the *certificate-based* checking. Sullivan, Wilson and Masson checked the *disjoint set union (DSU)* and a simplified priority queue that doesn't support *delete* or *change_key* in [15]. They also checked *making convex hull (CH)*, *sorting*, and *single-source shortest paths (SSP)* in [16]. Bright and Sullivan checked the full *priority queues (PQ)* supporting *delete* and *change_key* and the *mergeable priority queues (MPQ)* [4]. These certificate-based checkers are offline because they don't report errors immediately but rather maintain a query-result sequence as a certificate trail and verify the trail periodically to find errors. (Note that the hash-based checkers are also offline.) Finkler and Mehlhorn gave a different certificate-based checker for priority queues [9] with the same time and space efficiencies as the one in [4]. This checker is also offline but uses a trail other than the sequence of query-result pairs. Online certificate-based checkers maintain no trail but verify an individual certificate immediately after each operation. Such checkers are developed by Bright and Sullivan for *search data structures (SDS)*, *splittable search data structures (SSDS)*, and *nearest neighbor queries (NN)* [5]. All of the above certificate-based checkers, online or offline, take $O(n)$ (linear) space and $O(1)$ time per operation. The online checkers commit storage and operation invasions, while the offline checkers commit storage invasions only. There was no clue of realizing a certificate-based checker in sublinear space.

1.3   Our Contribution

Inspired by both methods of the hash-based and the certificate-based checking, we suggest a novel approach to checking value-sensitive data structures in sublinear space. We argue that the correctness of value-sensitive data structures consists of two components: *integrity* and *validity*. Then we observe that the hash-based checking technique checks not only the correctness of value-insensitive data structures but also the integrity of value-sensitive data structures. Next, we propose the concept of self-certification of a (value-sensitive) data structure, which is an augmented implementation of the data structure such that the result of an operation is self-certified to guarantee the validity. Using the self-certification and the hash-based techniques together, we realize a checker for SDS (search data structures), a fundamental value-sensitive data structure, which takes $O(\log n)$ space and checks each operation in $O(1)$ time w.h.p. Although the self-certification for SDS is quite simple, this may not be the case for other value-sensitive data structures. Thus, applying our method onto other value-sensitive data structures is non-trivial. However, the frame of our approach does isolate the process of self-certification as the only open part that needs to

Table 1. Comparison of different checking techniques

	Technique	Space	Time	Invasions
1	hash-based	$O(\log(n))$	$O(1)$ w.h.p.	storage, operation
2	online certificate-based	$O(n)$	$O(1)$	storage, operation
3	offline certificate-based	$O(n)$	$O(1)$	storage
4	hash + self-certification*	$O(\log(n))$	$O(1)$ w.h.p.	storage, operation

∗: New in this paper.

Table 2. Data structures that have been checked

	Technique	Applicable to
1	hash-based	RAM, stack, queue, linked structures
2	online certificate-based	SDS, SSDS, NN, PQ
3	offline certificate-based	DSU, PQ, MPQ, CH, sorting, SSP
4	hash + self-certification*	SDS*

∗: New in this paper.

be designed individually for each data structure or each ADT (abstract data type) operation. Therefore, we expect to see more sublinear space checkers of value-sensitive data structures inspired by our work. A comparison of our result and the previous results is shown in Table 1 and 2.

2   Our Idea

2.1   Describing Value-Sensitive Data Structures

We redefine any data structure D as a triple $D = (E, P, R)$, where E, P, R are the set of *elements*, *operations* and *rules*. Here we use the rules to describe all value-sensitive properties of the operations. (Therefore, a value-insensitive data structure is just a data structure with empty set of rules.) For example, a priority queue can be defined as (E, P, R) where

- each data element $e \in E$ stores a key value x;
- P includes operations of $insert(x)$, $delete(x)$, $change\_key(x, x')$, min, and $extract\_min$; and
- R contains one rule: "the element accessed by min or $extract\_min$ has the minimum key value among all key values stored in E".

That is, we consider a data structure as a repository of data elements (a bag of balls) that users can check out and check in elements via some operations following some rules. Observe that the only access or modification an operation can make to E is to check out or check in an element, defined as the following two *basic operations*:

- $put(e)$: check in an element e into E, i.e., change the status from $e \notin E$ to $e \in E$.
- $get(e)$: check out an element e from E, i.e., change the status from $e \in E$ to $e \notin E$.

Any operation is described as the combination of *put* and *get* plus some rules to follow. For example, the *change_key* operation in the above priority queue consists of a $get(e)$ and a $put(e')$; and the min operation consists of a $get(e)$ and a $put(e)$ following the rule that e contains the minimum value in E. Note that, as showed with the min, even if we just want to take a look at an element, we need to get it from E then put the same element back into E. In order to distinguish the two *basic operations get* and *put* and the original operations provided to users by the checkee, we call the original operations *data operations*.

2.2   Integrity and Validity

The correctness of an operation then bears two meanings: *integrity* and *validity*, which we describe in the following.

- Integrity refers to the authentic execution of each *get* and *put*. That is, a) $e \in E$ is "true" before the execution of $get(e)$ and becomes "false" after it; and b) $e \in E$ is "false" before the execution of $put(e)$ and becomes "true" after it.
- Validity of an operation means that all rules associated to this operation are followed, such as min indeed gets the minimum value.

In another word, the integrity is to get or put a ball in the bag honestly, and the validity is to pick the right ball. The data structure D functions correctly if and only if the integrity and the validity are both guaranteed.

2.3   Checking Integrity and Validity Separately

Recall that we define the *put* and *get* as *basic operations* and the original operations supported by the data structure D as *data operations*. Given a data structure D (the checkee), we follow the encapsulation model to design a checker and place it in between of users and D. The checker audits the queries and answers communicated between users and D and reports errors. It maintains a (E, P, R) description of D and checks the integrity and the validity in two separate components.

   The validity is checked online. The checker appends additional data operations to a user's data operation and sends them together to the checkee. Based on the results of the additional operations, validity of the user's operation can be verified. Here we must augment D in advance to make it capable of certifying the validity of one data operation with the results of some other data operations. We call this technique the *self-certification* of D, which we will illustrate with SDS in Section 3. Since the validity checking is done online and there is no need to keep a trail, this part of the checker takes constant space. Furthermore, the self-certification of SDS in this paper takes $O(1)$ storage invasions and the additional data operations appended to each user's data operation are done in $O(1)$ time at the checkee. It remains open to do self-certification for other value-sensitive data structures (such as priority queues) with $O(1)$ storage invasions and $O(1)$ time overhead.

In order to check the integrity, the checker transforms every data operation it sends to the checkee (including those of users and those appended by the checker itself) into basic operations and maintains a transcript of the data operations in the form of a sequence of basic operations. It then checks the integrity offline, i.e., to ensure that the basic operations recorded in the transcript are executed honestly at D, by using Blum *et al.*'s hash-based method in [3]. (See Section 3.3.) Since the transcript can be encoded in $O(\log n)$ space using the ϵ-biased hash functions, this part of the checker takes $O(\log n)$ space. Note that a data operation with constant size of input and output will be transformed into the combination of constant number of basic operations, regardless of the running time of that data operation at D. Therefore this integrity checking process brings $O(1)$ time overhead to any data structure that is already self-certificated.

The model of our checking scheme is showed in Figure 1. Next we'll use SDS as a concrete example to demonstrate how to design sublinear space checkers for value-sensitive data structures using our scheme. In Section 3.2 we show how to do self-certification and online validity checking for SDS. This part is not a uniform procedure for all data structures. It must be individually designed for each data structure. For many value-sensitive data structures, it could be a challenging task to do self-certification with non-excessive invasions. This is why our work is not a complete solution to Blum *et al.*'s open problem but only a feasible approach. In Section 3.3 we cite the method provided in [14,3] to show how to check the integrity of a sequence of *put* and *get* operations. This part is uniform for the integrity checking of all data structures. The two parts together give a complete checker for SDS, as well as an approach to checking other value-sensitive data structures.

3   Checking Search Data Structures (SDS) in Log Space

3.1   The Search Data Structures (SDS)

SDS=(E, P, R) is defined as follows. Each element in E contains a key value, and the values are comparable according to a total order. P includes the operations $insert(x)$, $delete(x)$, $search(x)$ $predecesor(x)$, $succesor(x)$, min and max.[2] Rules in R are summarized into two groups in the following.

1. Operation $search(x)$ returns an element $e \in E$, if x is the key value e; or returns "not exist" if x is not the key value of any $e \in E$.
2. Key values of the elements returned by $predecessor(x)$, $successor(x)$, min, and max follow the sorted order of all values stored in E. (Predecessor of the minimum and successor of the maximum are "null".)

---

[2] Some description to (a succinct version of) SDS contains only three operations $insert(x)$, $delete(x)$, and $search(x)$. However, it is straightforward to augment any implementation of the succinct SDS to support the rest operations in the context by using $O(1)$ size storage invasions and $O(1)$ time operation invasions. Therefore any checker of the (full version) SDS in this paper also checks the succinct version SDS with the same efficiencies and invasions, under Blum *et al.*'s model.
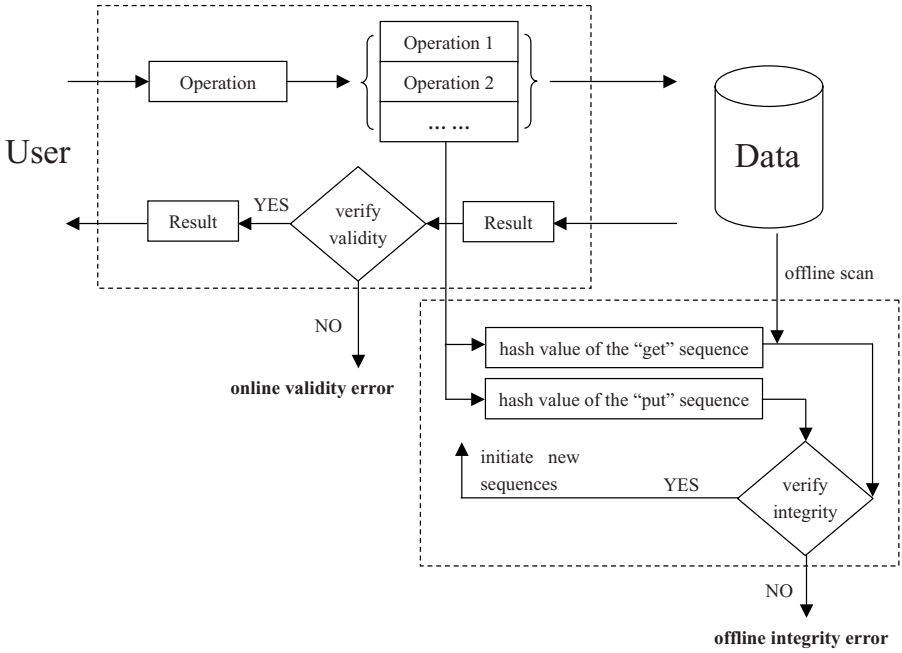
Fig. 1. The scheme of checking value-sensitive data structures

In one word, SDS organizes a set of values in sorted order and supports order preserving queries and updates. An efficient SDS would take $O(n)$ space for storage, $O(\log n)$ time to do each search, insert, or delete operation, and $O(1)$ time to do each of the other operations. A balanced binary search tree with modest augmentations to support a sorted linked list as well (details are omitted) is an example of efficient SDS implementations.

3.2   Self-certification and Validity Checking

We describe this process demonstrated with SDS.

Self-certification. Self-certification of a data structure $D = (E, P, R)$ includes the following three tasks.

1. The augmentation of each data element in E to associate additional information to the element via non-excessive storage invasions. Here for SDS we augment each element $e \in E$ to be $e = (x, predecessor(x), successor(x), i)$, where x is the key value stored in e. Here $predecessor(x)$ and $successor(x)$ associated to e are the key values of (not the links to) the predecessor and successor of x in E according to the sorted order, and i is an integer index indicating the order (discrete time) of insertion of x.

Any data operation that updates the SDS elements, such as an insertion or deletion, must be augmented accordingly as well in order to update the augmented data items consistently. For example, $insert(x)$ now needs to update the elements containing the values $predecessor(x)$ and $successor(x)$ after adding a new element containing x into E. This is like the process of inserting a new node into a linked list. The element containing $predecessor(x)$ must update its successor from $successor(x)$ to x, and the element containing $successor(x)$ must update its predecessor from $predecessor(x)$ to x. We omit the details of augmenting each SDS data operation.

2. A mapping from each data operation in P to a sequence of data operations in P. When auditing the user-checkee correspondence, the checker will substitute the corresponding sequence of operations for each user operations, and use the results of the sequence of operations as a certificate to verify the result of the user operation. Operation invasions caused by this mapping must be non-excessive, meaning that the time complexity of the sequence of operations at D must not exceed that of the original user operation. The mapping for SDS operations is showed in Table 3.

3. Algorithms for the checker to verify the result of each data operation with the results of the sequence of data operations it maps to. This verification checks the validity of the result of a user operation. (I.e., if there is no integrity error, then the operation result follows the rules in R correctly.) The verification algorithms for SDS are showed in the next.

Table 3. The operation mapping of SDS for validity checking

Data operation from user	Sequence of data operations sent to D
$insert(x)$	$predecessor(x), successor(x), insert(x)$
$delete(x)$	$predecessor(x), successor(x), delete(x)$
$search(x)$	$search(x), predecessor(x), successor(x)$
$predecessor(x)$	$predecessor(x), successor(x)$
$successor(x)$	$predecessor(x), successor(x)$
min	min
max	max

Verification algorithms. Algorithms showed below to verify the validity of SDS data operations are straightforward and involve only simple value comparisons. It is easy to justify the soundness of these algorithms, and to see that verifying each data operation takes $O(1)$ time and $O(1)$ space at the checker.

Recall that an element in E with key value x is $e = (x, y, z, i)$ where y and z are the predecessor and successor of x. To assist the presentation, we also denote by (x_1, y_1, z_1, i_1) the element returned by the operation $predecessor(x)$, and (x_2, y_2, z_2, i_2) the element returned by the operation $successor(x)$.

– Certifying $insert(x)$ or $delete(x)$ with $predecessor(x)$ and $successor(x)$. Checker verifies if $z_1 = x_2$, $y_2 = x_1$, and $x_1 < x < x_2$. Certifying $delete(x)$

is similar. Note that if one of the $predecessor(x)$ and $successor(x)$ is $null$, then verifying the other one still suffices the certification of validity. We omit the details of doing so. We also omit the details when x is already in E before the $insert(x)$ or when $x \notin E$ before the $delete(x)$, for which cases the verification is still necessary and the process of doing it is similar.

- Certifying $search(x)$ with $predecessor(x)$ and $successor(x)$.
  - If the search result is an element containing x, then the checker verifies if $z_1 = x = y_2$.
  - If it returns "not exist", then the checker verifies if $z_1 = x_2$, $y_2 = x_1$, and $x_1 < x < x_2$.

  Again, if one of the $predecessor(x)$ and $successor(x)$ is $null$, then verifying the other one is sufficient. Details are omitted.
- Certifying $predecessor(x)$ with $successor(x)$ or certifying $successor(x)$ with $predecessor(x)$. Checker verifies either $z_1 = x = y_2$ or ($z_1 = x_2$, $y_2 = x_1$, and $x_1 < x < x_2$). We again omit the cases when $predecessor(x)$ or $successor(x)$ is $null$.
- Certifying min or max by itself. Simply verify if the predecessor value of the assumed min is $null$, or the successor value of the assumed max is $null$.

The above algorithms yield the following result. We omit the proof in this preliminary version.

Theorem 1. *We can check the validity of each SDS data operation in $O(1)$ time and $O(1)$ space, by introducing storage invasions of $O(1)$ size per data element and operation invasions of $O(1)$ time per user operation, which are minimal.*

3.3   Checking Integrity with Blum *et al.*'s Method

Properties of ϵ-biased hash functions were studied and fast algorithm to update the hash values was provided by J. Naor and M. Naor in [14]. Its application in checking value-insensitive data structures was discovered by Blum *et al.* in [3]. In fact this technique checks not only the value-insensitive data structures but the integrity of any data structures. (The reason it works solely for checking value-insensitive data structures is that these data structures have empty rules so there is no validity to check.) We present the method in [3] of using hash-based technique to check integrity in the following.

- Checker maintains an integer timer t that increases by 1 after each put, which determines the index i of an element $e = (x, y, z, i)$ that is put into E.
- Checker maintains two transcripts W and R to record respectively the sequence of elements that have been put into E and the sequence of elements that have been got from E. Checker transforms every data operation sent to the checkee (including the user's operations and the checker's invasive operations appended to it for validity checking) into the form of put and get, and appends them to the sequences W and R accordingly.

- If an element e is got out of E and put back later, the time stamp i of it must be updated to the current time t. In another word, the element put back to E is not the one that was got out but a new element. This means that each element is involved in exactly one *put* and one *get* in its life cycle. Elements that are still contained in E have not experienced the *get*.
- Checker periodically scans E to get all elements out. Thus during this period of time the transcripts R for the sequence of *get* and W for the sequence of *put* should be identical. Comparing the two transcripts is sufficient to verify the integrity of the execution of the basic operations *put* and *get* during the past period. (The elements scanned are put back to E after the verification, and those *put* operations are recorded in a new transcript W starting with a new period of integrity checking.)
- Instead of maintaining R and W (each of length $O(n)$) explicitly, the checker maintains the description of an ϵ-biased hash function h and two hash values $h(R)$ and $h(W)$ as fingerprints of R and W. This takes $O(\log n + k)$ space with a constant parameter k. Whenever a new *get* or *put* is appended to R or W, the checker updates $h(R)$ or $h(W)$ accordingly. The amazing properties of ϵ-biased hash functions allow updates to be done bit by bit, without re-hashing the hash value, taking only $O(k)$ time to record a *get* or *put*. All together, the integrity checker uses $O(\log n)$ space and linear time (amortized $O(1)$ time per operation) with a constant parameter k, and reports integrity errors offline (periodically) with probability $1 - 2^{-k}$.

We omit further details of how this method works. Interested readers are referred to read [14,3]. The result of integrity checking is in the following.

Theorem 2. *[14,3] The integrity of any data structure of size $O(n)$ can be checked with probability $1 - 2^{-k}$, k is a constant, by a hash-based checker using $O(\log n)$ space and amortized $O(1)$ time per operation. Such an integrity checker commits $O(1)$ size storage invasions per data element and $O(1)$ time operation invasions per data operation, which are minimal.*

4   Conclusion

We have provided an approach to designing sublinear space checkers for value-sensitive data structures. The framework and the integrity checking component apply for all data structures. The validity checking component must be individually designed for each value-sensitive data structure. We've only showed as a demonstration how to do it on SDS. The two components in Section 3.2 and 3.3 together fulfill a log space, optimal time, and minimal invasion checker for SDS. We conclude with this result in the following theorem, which can be viewed as a step towards a positive answer to the open problem of Blum *et al.* in [3].

Theorem 3. *We can check a search data structure (SDS) with probability $1 - 2^{-k}$, k is a constant, in $O(\log n)$ space and amortized $O(1)$ time per operation, while committing only minimal storage and operation invasions.*

References

1. Amato, N.M., Loui, M.C.: Checking linked data structures. In: FTCS-24: 24th International Symposium on Fault Tolerant Computing, Austin, Texas, pp. 164–175. IEEE Computer Society Press, Los Alamitos (1994)
2. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proc. of the 2002 ACM Symp. on Principles of Database Systems (PODS 2002) (2002)
3. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica 12, 225–244 (1994)
4. Bright, J.D., Sullivan, G.: Checking mergeable priority queues. In: Digest of the 24th Symposium on Fault-Tolerant Computing, 1994, pp. 144–153. IEEE Computer Society Press, Los Alamitos (1994)
5. Bright, J.D., Sullivan, G.: On-line error monitoring for several data structures. In: Digest of the 25th Symposium on Fault-Tolerant Computing, 1995, pp. 392–401. IEEE Computer Society Press, Los Alamitos (1995)
6. Bright, J.D., Sullivan, G., Masson, G.M.: Checking the integrity of trees. In: Digest of the 25th Symposium on Fault-Tolerant Computing, 1995, pp. 402–411. IEEE Computer Society Press, Los Alamitos (1995)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT & McGraw-Hill (2001)
8. Devanbu, P., Stubblebine, S.: Stack and queue integrity on hostile platforms. IEEE Tran. Software Engineering 28(1), 100–108 (2002)
9. Finkler, U., Mehlhorn, K.: Checking priority queues. In: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, Baltimore, Maryland, pp. 901–902 (1999)
10. Goodrich, M.T., Tamassia, R.: Data Structures and Algorithms in Java, 2nd edn. Wiley, Chichester (2001)
11. Kratsch, D., McConnell, R., Mehlhorn, K., Spinrad, J.: Certifying algorithms for recognizing interval graphs. In: SODA 2003, pp. 158–167 (2003)
12. Mehlhorn, K., Näher, S.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (1999)
13. Muthukrishnan, S.: Data streams: algorithms and applications. Technical report, Rutger (2003)
14. Naor, J., Naor, M.: Small-bias probability spaces: Efficient constructions and applications. SIAM J. Comput. 22(4), 838–856 (1993)
15. Sullivan, G., Wilson, D., Masson, G.: Certification trails for data structures. In: Proceedings of the 21st Annual Symposium on Fault-Tolerant Computing, pp. 240–247 (1991)
16. Sullivan, G., Wilson, D., Masson, G.M.: Certification of computational results. IEEE Transactions on Computers 44(7), 833–847 (1995)
17. Wasserman, H., Blum, M.: Software reliability via run-time result-checking. Journal of the ACM 44(6), 826–849 (1997)