A Composite Stabilizing Data Structure[⋆]

Ted Herman and Imran Pirwani

Department of Computer Science, University of Iowa
{herman,pirwani}@cs.uiowa.edu

Abstract. A data structure is stabilizing if, for any arbitrary (and possibly illegitimate) initial state, any sequence of sufficiently many operations brings the data structure to a legitimate state. A data structure is available if, for any arbitrary state, the effect of any operation on the structure is consistent with the operation's response. This paper presents an available stabilizing data structure made from two constituents, a heap and a search tree. These constituents are themselves available and stabilizing data structures described in previous papers. Each item of the composite data structure is a pair (key,value), which allows items to be removed by either minimum value (via the heap) or by key (via the search tree) in logarithmic time. This is the first research to address the problem of constructing larger data structures from smaller ones that have desired availability and stabilization properties.

1 Introduction

Availability is an important topic in online system design. Ideally, a system should respond to requests in a timely manner in spite of hardware failures, bursts in load, internal reconfigurations, and other disruptive factors. The usual technique for ensuring availability is to engineer a system with sufficient redundancy to overcome failures and resource shortages [13, 11, 5].

One attraction of self-stabilization is that it does not require the traditional type of resource redundancy to deal with faults. The question then comes to mind, can self-stabilization be enhanced to support system availability? We begin to address this question with a low-level task, which is to consider data structures. Since data structures are frequently used in software for systems, the key question is: can data structures that support availability in spite of transient failures be constructed? The answer is not obvious, since most operations on data structures either abort, get caught in a loop, throw exceptions, or have unpredictable behavior when internal variables have invalid values. Specifically, we study one data structure in this paper, a composite data structure made from a heap and a search tree. This data structure is of interest because it shows how one available data structure can be constructed from smaller, available data structures (of course, general compositional methods are the ultimate goal, but examples are helpful to understand the technical difficulties).

⋆ This research is sponsored by NSF award CAREER 97-9953 and and DARPA contract F33615-01-C-1901.

The literature of self-stabilization differs from our treatment in several ways. First, we do not consider a distributed system, which is the normal model for self-stabilization [3]. Second, most self-stabilizing algorithms make no guarantees about behavior before the system reaches a legitimate state — whereas we require some guarantees for availability. Third, the data structure model of operations in this paper restricts transient faults to effect only the variables of the data structure and not the internal variables of an operation underway (in principle, our results could be extended to allow such corruption as well). Our data structure results are stabilizing in the following sense. Initially, the content of a data structure may be arbitrary and corrupt. During some initial sequence of operations on the data structure, the response time of each of these operations could be abnormally large, but not larger than that for a legitimately full data structure; and some operations during this sequence could respond with errors, such as reporting that an item could not be inserted, even though the data structure is not full. In all cases, however, the response is consistent with how the operation changed the data structure. Finally, after a sufficiently long sequence of operations, the data structure's state is legitimate, and thereafter all operations have normal running times and responses.

While there are numerous studies of fault-tolerant data structures, the fault model for these studies does not consider recovery from unlimited transient faults in the data structure. Self-stabilization is required to deal with unlimited transient faults, yet very few papers in the area of self-stabilization treat data structures in the model of operations applied to the structures. Two works [12, 10] mention self-stabilization the context of objects that undergo operations before a legitimate state is reached. The challenge of [10] is that operations are concurrent and wait-free, which is an issue beyond the scope of our present investigation. The notion of an available and stabilizing data structure is new in [8], which presents an available, stabilizing, binary heap. An available and stabilizing 2-3 tree is described in [9].

The data structures investigated in this paper and related papers [8, 9] are fixed-capacity structures. The reason for fixing the capacity is to include the design of dynamic storage allocation in the implementation. Standard texts presenting data structures may gloss over the role of allocation in dynamic structures, but for the case of stabilization, an incorrect initial state can make the storage allocation pool appear empty even though the data structure contains very few items. The situation is even more complex when numerous data structures share a common allocation pool. At this stage of research, we investigate single data structures, hence the constraint of fixed capacity to make the presentation self-contained. Also, the data structure operations considered here are single-item operations (insert, delete, find); operations such as set intersection or union are not investigated. Many research directions thus remain unexplored, including inter-structure operations, data structures of unlimited capacity, and shared allocation pools (and of course, the question of concurrent operations).

Organization of Paper. Section 2 describes the model of data structures and operations, and then defines the availability and stabilization precisely. Section 3 presents a simple example of an data structure and briefly reviews the constructions for heap and search tree. Section 4 defines the main problem we consider, and sketches an impossibility result related to this problem. Section 5 gives an overview of the main construction. Detailed correctness proofs of the construction are omitted from this extended abstract. The paper ends with discussion in Section 6.

2 Availability, Stabilization, and Operations

A data structure is an object containing items. The object is manipulated by a fixed set of operations. Each type of operation is defined by a signature (operation name and invocation parameters) and a set of responses for that signature. The relation between a signature and its response is specified by sets of sequential histories of operations applied to the data structure. That is, we specify operation semantics by a collection of legal sequences rather than by presenting pseudocode. We do this in order to maximize the freedom of the object's implementor to choose data representation and algorithms.

A history is an infinite sequence of pairs $\langle (op_1\ res_1)\ (op_2\ res_2)\ \cdots \rangle$ where op_i is the i-th operation invocation (including its parameters) and res_i is the response to op_i. A *point* in a history refers to the state of the data structure either before any operation or between two operations op_i and op_{i+1}. The *content* of the data structure at a point t is defined directly if t is the initial point of the history; if t is not the initial point, then the content is defined in terms of the sequence of operations and responses leading up to point t. Let C_t denote the content at point t. Let $|C_t|$ denote the number of items in the data structure at point t. All data structures have a fixed *capacity*, which is an upper bound on the number of items the data structure is allowed to contain.

We illustrate content specification by history with a small example, a data structure for a set of at most K items with `insert` and `remove` operations. Let the content of the set be empty at the initial point of any history. The content C_t at point t following operation op_t is defined recursively: if op_t is an `insert`(x), and $x \notin C_{t-1} \wedge |C_{t-1}| < K$, then $C_t = C_{t-1} \cup \{x\}$ and the response to the operation is *ack*; if op_t is an `insert`(x) and $|C_{t-1}| = K \vee x \in C_{t-1}$, then $C_t = C_{t-1}$ and the response to the operation is *full* if $|C_{t-1}| = K$ or *ack* otherwise. If op_t is a `remove`(x), then $C_t = C_{t-1} \setminus \{x\}$ and the response is *ack*.

Throughout the paper we assume the following for data structures and their operations. Operations are single object methods and we suppose that all operations in a history are operations on the same data structure (so the example above cannot be extended to allow set union and intersection operations, which combine objects). An operation op_t is called *moot* if $C_t = C_{t-1}$. We classify each operation op_t as either *successful* or *unsuccessful* depending on its response, and this classification is specified as part of the suite of operations for a data structure. In this classification, all unsuccessful operations are moot, but the converse need not hold.

The intuition for "unsuccessful" classification is that, although the operation is guaranteed to be moot, the response of the operation may not be trustworthy if the history begins with a data structure damaged by a transient fault. We formalize this below in the definitions of availability and stabilization. For the set example above, only the response *full* could be classified as an unsuccessful operation; if *full* is an unsuccessful response, then intuitively, in a damaged state, an `insert` operation that responds *full* is allowed to do so even though the set contains fewer than K items.

A *well-formed* history is one where the responses of all operations agree with the definition of content for the data structure. A detailed specification of what is means for a history to be well-formed depends on the semantics for the data structure operations, their responses, and the definition of content. A *legitimate* history is a well-formed history so that the operation running times are within the bounds, as a function of content, given by the implementation for that data structure. For example, suppose the set data structure has an implementation where the running time for an operation on a set of size m is linear in m. An example of an illegitimate history is one where an `insert`(x) operation fails at some point t even though $|C_t| < K$. Another example of illegitimacy is a `remove`(x) operation that has 2^m running time at point t although $|C_t| = m$. Observe that if all moot operations are removed from a legitimate history, the result is either a legitimate history or a (finite) prefix of a legitimate history.

Given history H, let H_t denote the suffix of H following point t. Let $U(I)$, for a history I or segment I of a history, be the sequence obtained by removing all unsuccessful operations from I. A history H is *available* if there exists a point t and a sequence of operation invocations P such that $P \circ U(H_t)$ is a well-formed history or a (possibly empty) finite segment of a well-formed history ($\circ$ is the sequence catenation operator); also, the running time of any operation in H is no more than the worst-case running time of any operation, taken over all legitimate histories (usually this worst-case running time is obtained for an operation on a full data structure). Because unsuccessful operations are moot, it follows that legitimate histories are available histories. Examples of histories that are available but not legitimate include histories with operations whose running times larger than one would expect for the content, and operations that return unsuccessful responses in unexpected cases (e.g. an `insert`(x) responds *full* even though the content has less than the data structure's capacity number of items). An implementation of a data structure is available iff all its histories are available. A trivial implementation of a set to guarantee availability would be to have any `remove` do nothing to the data structure and respond *ack* in $O(1)$ time, and to have any `insert` do nothing to the data structure and respond *full* in $O(1)$ time. To see that this implementation is available, take any history H, let t be the initial point of H, let P be empty, and choose C_t to be the empty set. Then $P \circ U(H_t)$ is a legitimate history since it has no `insert` operations and `remove` operations are moot.

A history H is *stabilizing* if there exists a point t and a sequence of operation invocations P such that $P \circ H_t$ is a legitimate history. In the definition of a stabilizing history, there can be many choices for the point t and prefix P to make $P \circ H_t$ a legitimate history; call t the *stabilization point* if there exists no point s preceding t such that $Q \circ H_s$ is a legitimate history for any prefix Q. An implementation of a data structure is stabilizing iff all its histories are stabilizing. The *stabilization time* of a stabilizing history H is the number of operations preceding its stabilization point. The stabilization time of an implementation of a data structure is the maximum stabilization time of all of its histories (if there is no maximum, then the stabilization time is infinite).

3 Available and Stabilizing Components

Before presentation of the composite data structure in Section 4, we review first the constituent data structures used to build the composite. Some constituents are trivially available and stabilizing: for instance, we may regard an atomic variable (word of memory) to be a data structure supporting `read` and `write` operations. It is simple to show that such variables are available and stabilizing. Interesting data structures such as heaps and search trees require nontrivial constructions to satisfy availability and stabilization. Between the trivial case of a variable and more advanced data structures, we consider first in this section two elementary data structures, a queue and a type of stack, to illustrate some challenges of implementing availability and stabilization.

3.1 Queues, Stacks, and Conservative Implementations

Two elementary examples of available and stabilizing data structures are described here, a queue and a type of stack. We then show two variations on the queue that do not have available and stabilizing implementations. These variations on the queue are somewhat artificial, but they are useful to illustrate difficulties that arise later in the presentation of a composite data structure. An important concept introduced in this section is a *conservative* implementation of a data structure. Informally, an implementation is conservative if operations cannot substitute values for missing data.

A *K-queue* is a queue with a capacity of K items supporting `enqueue` and `dequeue` operations. The response to an `enqueue`(x) operation is either *ack* or *full*, and the response to a `dequeue` is either an item or *empty*. Only the *full* response is classified as an unsuccessful response. The content C_t at any point t can be defined from the sequence of successful `enqueue` operations up to point t and the sequence of `dequeue` operations that return items up to point t (we omit the formal definition). In any legitimate history, the running time of an operation is $O(1)$. In any well-formed history, responses have the following properties. An `enqueue` operation at point t responds *full* iff $|C_t| \geq K$, and otherwise responds *ack*; a `dequeue` at point t responds *empty* iff $|C_t| = 0$ and otherwise responds with the initial item of sequence C_t.

The conventional implementation of a bounded queue by a circular array is an available and stabilizing K-queue. Let `A`$[0..(K-1)]$ be an array of K

items and let `head` and `tail` be two integer variables. The K-queue is empty if ($\texttt{head} \bmod K$) = ($\texttt{tail} \bmod K$), and full if (($\texttt{tail} + 1) \bmod K$) = ($\texttt{head} \bmod K$). An $\texttt{enqueue}(x)$ succeeds iff the K-queue is non-full by writing x to $\texttt{A}[(\texttt{tail}+1) \bmod K]$ and then assigning $\texttt{tail} \leftarrow (\texttt{tail} + 1) \bmod K$ (the capacity of this queue representation is thus $K-1$ items). The `dequeue` is similarly defined, with the result that the state of the K-queue is well-defined for all possible values of `head`, `tail`, and `A`. By straightforward expansion of the definitions, availability and stabilization hold for this implementation and the stabilization time is zero.

The composite data structure described in Section 5 makes use of a relative of the K-queue. A *lossy K-stack* is a stack with a capacity of K items and `push` and `pop` operations. A `push` operation for the lossy K-stack always succeeds and a `pop` operation always returns an item. The K-queue can be implemented by an array of K elements and a `top` variable to contain an index for the top of the stack, which is incremented modulo K by `push` and decremented modulo K by `pop`. This stack is "lossy" because, for $K + 1$ consecutive `push` operations, the last `push` writes over the oldest item on the stack. We omit the straightforward details of the formal definition of this data structure and verification of its availability and stabilization properties.

The elementary examples of queue and stack become more challenging when there are domain restrictions on the items contained in the data structure. Consider a queue that may only contain (pointers to) prime numbers as items. The `enqueue` operation can return a new response *err*: the response to $\texttt{enqueue}(x)$ is *err* for any nonprime x, and the operation is moot. The running time of `enqueue` is no longer constant, because `enqueue` tests its argument for primality. The `dequeue` operation has constant running time, either returning (a pointer to) an item or the *empty* response. Now suppose we have an implementation of this data structure and let σ be the state for some queue of prime numbers. It is possible for a transient fault to transform σ into some σ' by changing some (pointers to) items in the queue from prime to nonprime values. This is a problem because the `dequeue` operation can now return a nonprime number. Returning a nonprime number is not possible in a legitimate history and because an item in response to `dequeue` is a successful operation, there is a conflict with availability if `dequeue` is allowed to return a nonprime number. The only resolution is for `dequeue` to check an item for primality before responding. We do not know of any method to test primality in constant time, so `dequeue` will require nonconstant running time for faulty states such as σ'. If the implementation is also stabilizing, then every history has a suffix where all `dequeue` operations have constant running time. Yet it is impossible for `dequeue` to distinguish between correct and faulty states in constant time unless there is a constant-time primality test. Therefore, every `dequeue` operation on a nonempty queue checks for primality, which conflicts with the running time constraint for a stabilizing implementation. We have thus sketched the proof of the following.

Lemma 1. No implementation of the prime queue is available and stabilizing.

Another difficulty with a domain-restricted queue is shown by a queue that may only items of the form *a:b* with a an even number. Again, execution of `enqueue`(*x:y*) reponds with *err* for odd x, but this can be checked in constant time. The `dequeue` operation can also verify that the head item has an even first component in constant time, so all operations run in constant time for a legitimate history. After a transient fault, there could be queue items with odd first components, so the question arises, what should `dequeue` do if it detects that the head is odd? One choice would be to respond with *0:z* in place of any item *r:z* with odd r found at the head of the queue. This choice would satisfy availability, because there is a legitimate history where zero is enqueued in place of any odd value (notice this method cannot be used for the prime number queue because of running time constraints). If there are further domain restrictions on items, say a relation between a and b in a pair *a:b*, or perhaps relationships with other variables outside the queue, then the substitution of *0:z* for *r:z* would not be valid.

We call an implementation *conservative* if no operation returning a data structure item is allowed to invent the item — a conservative implementation can only return items that are present in the data structure, and cannot coerce invalid values to legal ones. This is a key decision in our presentation, and differs from classical work on self-stabilizing control structures. For classical problems such as mutual exclusion, it does not matter how illegitimate states are converted into legitimate ones, but for the stabilization of data structures, we prefer to limit techniques to conservative measures where data items are not created to satisfy a domain constraint. The motivation for conservative operations is not just to make a theoretical problem nontrivial: in practice, stabilizing algorithms that limit the effect of faults are preferable to those that do not enforce any such limit [4, 6, 7]. While it is true that a transient fault could inject apparently legal values never actually inserted in a data structure, such transient faults are uncontrolled, whereas operation implementation can be designed to avoid the injection of artificial values (moreover, practical techniques such as error detecting codes can decrease the probability of legal values injected by faults).

Lemma 2. No conservative implementation of the *x:y* queue is available and stabilizing.

The lemma can be shown by an adversary argument: each `dequeue` has to distinguish, in $O(1)$ time, whether the queue is empty or has an item meeting the domain constraint, and for whatever strategy `dequeue` uses to examine the queue, an initial state can be constructed to defeat that strategy.

3.2 Heap and Search Tree Review

The heap and search tree data structures are more complicated than the queue, and items in the heap and search tree have domain restrictions, unlike a simple queue. In the heap, an item's value is the least of a subtree of values, and in a search tree, item keys are ordered. There is, however, a crucial way in which these domain restrictions are simpler than the example of the prime number queue: there is an implementation so that all operations access items via a path from the tree's root. This fact generalizes to the observation that, for any values of items in a such a tree, there is a maximal subtree for which the domain restrictions hold. For the heap this is called the *active heap* and for the search tree this is called the *active tree*.

The available and stabilizing search tree given in [9] is a 2-3 tree. Its active tree is defined as the maximal subtree so that the distance from root to leaf is the same for all leaves, all keys are in order, and several other validity conditions hold for child and parent pointers. For the available and stabilizing heap [8], the active heap is taken to be the maximal rooted subtree so that each node's value is a lower bound on the values of its children. The table below shows the signatures and responses for the 2-3 tree and heap structures. The method to ensure availability is straightforward: all data structure operations are implemented with respect to the active structure. A consequence of this method is that the active structure can be unbalanced, so that operations on the active tree may have no longer have running that is logarithmic in the number of active items. The remaining implementation task is stabilization, which entails balancing the active structure. Informally, the balancing of the active structure is a "background activity" similar to garbage collection in memory allocation schemes. This background activity is also needed to repair pointers, repair free storage chains, and correct various other internal variables of the data structure. Because no actual background process is assumed by the model of data structures, each operation on the data structure invokes a limited amount of background processing. The running time of any call to background processing is $O(\lg K)$ in [8,9] in an arbitrary state and $O(\lg n)$ after the stabilization point, where K is the capacity of the data structure and n is the number of items in the active structure. These running times are the same as the operation complexities, since in the worst case, an active heap or search tree encounters a path of length $O(\lg K)$, and when the structure is balanced, all paths are $O(\lg n)$.

signature	successful	unsuccessful	illegitimate	legitimate
`STinsert`(k, d)	*ack*	*full*	$O(\lg K)$	$O(\lg n)$
`STdelete`(k)	*ack*		$O(\lg K)$	$O(\lg n)$
`STfind`(k)	$(k, d)/$*missing*		$O(\lg K)$	$O(\lg n)$
`Hinsert`(v, e)	*ack*	*full*	$O(\lg K)$	$O(\lg n)$
`Hdeletemin`$(\ )$	$(v, e)/$*empty*		$O(\lg K)$	$O(\lg n)$
`Hdelete`(p)	*ack*		$O(\lg K)$	$O(\lg n)$

4 Composite Data Structure

The composite data structure presented here is called the *heap-search* tree. Definitions of the heap-search operations are explained in this section; the construction of the heap-search tree is presented in Section 5. The table below presents our initial table of the signatures and responses of the operations; later in this section we revise this table, after presenting an impossibility result for a conservative implementation.

signature	successful	unsuccessful	illegitimate	legitimate
`insert`(k, v)	*ack*	*full*	$O(\lg K)$	$O(\lg n)$
`delete`(k)	*ack*		$O(\lg K)$	$O(\lg n)$
`find`(k)	$(k, v)/$*missing*		$O(\lg K)$	$O(\lg n)$
`deletemin`( )	$(k, v)/$*empty*		$O(\lg K)$	$O(\lg n)$

Each item of the heap-search tree is a pair (k, v), and the content of the heap-search tree at any point is a multiset (bag) of such pairs. Below we use union ($\cup$) and subtraction ($\setminus$) for multiset operations, so $C \setminus \{(k, v)\}$ removes at most one copy of (k, v) from multiset C. Let s and t be consecutive points in a legitimate history and let the operation and response occur between s and t. Operation `insert`(k, v) is moot and responds *full* if $|C_s| \geq K$; otherwise $|C_s| < K$ and `insert`(k, v) responds *ack*, with $C_t = C_s \cup \{(k, v)\}$. Operation `delete`$(k)$ responds with *ack*; the operation is moot if there exists no b satisfying $(k, v) \in C_s$, otherwise $C_t = C_s \setminus \{(k, v)\}$ for some v satisfying $(k, v) \in C_s$. Operation `find`(k) is always moot, and either returns a pair (k, v) for some b satisfying $(k, v) \in C_s$, or returns *missing* if no such pair exists. Operation `deletemin`( ) returns *empty* and is moot if $|C_s| = 0$, otherwise the response is a pair (k, v) such that v is the minimal value for any pair's second component, and $C_t = C_s \setminus \{(k, v)\}$ in this case.

Before making statements about heap-search tree implementations, we first formalize what it means for an implementation of this data structure to be a composite of the heap and 2-3 tree. Informally, the implementation is a composite if it is a construction made by assembling one heap and one 2-3 tree, so that any pair $(k, v) \in C_s$ is represented by the item k in the 2-3 tree at point s and v in the heap at point s, and no other data structures are used in the storage of items. More formally, the implementation of the heap-search tree satisfies the following three constraints: (i) the composite heap-search tree implementation has one 2-3 tree S, one heap T, and possibly other data structures used for background processing; (ii) at any point s in a history, $(k, v) \in C_s$ iff $(k, d) \in S_s$ and $(v, e) \in T_s$, where S_s and T_s respectively denote the contents of the 2-3 tree and heap at point s, with d and e being associated data as defined by operations; and (iii) for any $(k, v) \in C_s$, the key k is contained only in the 2-3 tree — keys are not contained in any structure other than the 2-3 tree; similarly, the value v is not contained in any other structure than the heap. Note that (iii) could be ambiguous for a structure with integer keys as well as integer heap values, because a key could coincidentally reproduce a heap value. To resolve this ambiguity, assume that keys and heap values are taken from different types (say key and value) for purposes of defining constraint (iii).

Lemma 3. No conservative implementation of the composite heap-search tree using the heap and 2-3 tree is available and stabilizing.

The conclusion we draw from Lemma 3 is that either we should settle for an implementation that is not conservative, or the operations should be modified. The proof of Lemma 3 (omitted from this extended abstract) points out that `deletemin` is responsible for the difficulty, so we propose the following change: let `deletemin` have a new response *err*, to be returned if the value returned by `Hdeletemin` does not have a corresponding key in the 2-3 tree. This change satisfies availability by classifying any `deletemin` with an *err* response as an unsuccessful operation. However to satisfy stabilization, we shall require that no `deletemin` be unsuccessful after the stabilization point, since the *err* response does not occur in a legitimate history. For application purposes, *err* is useful because it informs the `deletemin` caller that something is incorrect in the heap-search data structure, but by returning within $O(\lg K)$ time, quickly gives the application the choice of repeating `deletemin` (and progressing toward stabilization) or using some other type of recovery. Lemma 3 can also be proved using the `find` operation instead of `deletemin`, because the search tree may contain duplicate keys in an initial state; the `delete` has another similar difficulty. Therefore, for the remainder of the paper, let `delete`, `find`, and `deletemin` return *err* if the operation is unsuccessful.

5 Heap and 2-3 Tree Composite Construction

5.1 Variables, Constituent Structures, and Pointers

The composite data structure is composed of two binary variables `STbit` and `Hbit`, a variable `curcolor` with domain {0,1,2}, a K-lossy stack, a heap, and a 2-3 tree. As explained earlier, all of these constituents (variables and structures) are available and stabilizing components. Throughout the remainder of the paper, K is the capacity of the heap-search tree and also the capacity of the heap and 2-3 tree. For convenience, we use the term `nextcolor` to mean $(1 + \texttt{curcolor}) \bmod 3$ and the term `prevcolor` to mean $(2 + \texttt{curcolor}) \bmod 3$.

Our construction uses pointers to connect heap items and 2-3 tree items. The use of pointers in conventional random access memory is challenging because damaged pointers can lead to further damage in data structures (for instance, modifying data accessed by a damaged pointer). We make some assumptions concerning how data, especially the heap and 2-3 tree, are arranged in memory. We assume that a procedure `Hpointer`(p) evaluates p and in $O(1)$ time returns *true* if p could be an item of the heap, and *false* otherwise. An implementation of `Hpointer`(p) could check that p is an address within a range $[Hstart, Hend]$ defined for heap items, and also check that p is a properly aligned address ($Hstart$ and $Hend$ are program constants not subject to transient fault damage). Similarly, we assume there is a procedure `STpointer`(p) to determine whether p can be a 2-3 tree item.

The next level of pointer checking is to determine whether or not a given p is a pointer to an item in the active structure. Let `STintree`(p) respond *true* if p is a pointer to an item in the active 2-3 tree, and *false* otherwise. The running time of `STintree`(p) is $O(h)$ where h is the height of the active 2-3 tree. The implementation of `STintree`(p) could be similar to one described in [9]; after using `STpointer`(p) to validate candidacy of p, the procedure follows parent pointers of tree items to verify that the tree's root is an ancestor, and also verifies properties of keys in items are such that all items in the path are in the active 2-3 tree. A similar procedure `Hinheap`(p) determines whether p is an item of the active heap. `Hinheap`(p) can be implemented by tracing the parentage of p back to the heap's root. Unlike `STintree`(p), which consumes $O(h)$ time, the complexity of following p's parentage for arbitrary p satisfying `Hpointer`(p) is $O(\lg K)$ — the running time is $O(h)$ if p is an item of the active heap. In some cases, the time bound for `Hinheap`(p) should be constrained, even if the result is inaccurate. Let `Hinheap`(p, t) be an implementation that limits the parentage trace to t iterations, and if t iterations do not suffice to reach the heap's root, `Hinheap`(p, t) returns *false*.

Each item of the K-lossy stack is a pointer. Items of the heap and 2-3 tree provide for a data field in the respective `insert` operations (see (k, d) and (v, e) in the table defining operations), and we use these data fields for pointers. For an item (k, d) of the 2-3 tree, d is a pointer. For an item (v, e) of the heap, $e = (q, c)$ with q being a pointer and c being a "color" in the range $\{0,1,2\}$. Our convention is to refer to d as the pointer associated with the 2-3 tree item (k, d), to call q the pointer associated with heap item (v, e), and to call c the color associated with (v, e). We also use the notation x`.color` to refer the color of an item x.

At any point in a legitimate history, the pointers associated with items in the active heap and active 2-3 tree should bind a pair (k, v), which means that the pointer d associated with k in the 2-3 tree refers to an item with value v in the heap, and in turn the pointer q associated with v refers to the item (k, d). Let `STcross`(p), for p a pointer to a 2-3 tree item (k, d), be a boolean function returning *true* only if `Hpointer`(d) is *true* there is a pointer q associated with the heap item of d, and $q = p$. A similar function `Hcross` is defined for heap items, and we generically use the term *crosscheck relation* to mean that the pointer associated with an item, heap or 2-3 tree, refers to an item in the other structure that has the expected back pointer; we say that two items crosscheck if they satisfy the crosscheck relation. Note that the crosscheck relation can be checked in $O(1)$ time, but satisfying the crosscheck relation does not imply membership in the active heap or 2-3 tree.

5.2 Modifications to Constituent Operations

We change the operations of the heap and 2-3 tree only by adding a some extra steps to look after the pointers and the color field associated with a heap item. The reason that pointers are an issue is that items can change location inside a data structure as a result of insertions or deletions. For the 2-3 tree, this can occur by node splitting or merging. For the heap, this occurs by item swaps as part of "heapify" routines to restore the heap property after an item is removed or added.

The change relating to pointers is: whenever an operation moves an item x, the operation validates x by a crosscheck, and if the crosscheck holds, then the operation adjusts pointers so that the crosscheck relation will be hold after the item moves. Conversely, if the crosscheck relation does not hold for x prior to the move, then the pointer is forcibly invalidated so that crosscheck will not hold (accidentally) as a result of moving x. This change applies to both heap and 2-3 tree operations, since procedures of both can move their respective items within the active structures. Background activities that move items are also changed to attend to item movement. In particular, the background heap activity includes an `balance` routine that deletes an item of maximum depth, reinserting it at minimum possible depth; the movement of this item by `balance` requires pointer adjustment so that subsequent crosschecks are valid.

The change relating to colors only applies to heap operations and background activities. Whenever an operation or background routine examines an active heap item with color c, it pushes a pointer to that item on the K-lossy stack if $c = \texttt{nextcolor}$, and then assigns the item's color to be the value of `curcolor`. A single operation or background routine for the heap may examine many nodes (for instance, examining all the nodes along a path from root to a leaf), so an operation can push many pointers on the stack as a result of this change. However, since checking the color field and stack pushes are $O(1)$ time steps, the operation complexities of the heap are unchanged.

Two background activities of the constituent structures have new duties in the composite data structure. First, we review terminology for the existing background operations. The heap has a background operation called `Hscan` and the 2-3 tree has a similar operation called `STscan`. One intent of these background operations is the same for both structures, which is to examine a path of nodes from root to leaf, possibly truncating nodes that are not part of the active structure, correcting variables within nodes, and in the case of the 2-3 tree, possibly merging nodes. One `Hscan` invocation can examine up to $\lg K$ items, since each node in the active heap contains an item. One `STscan` invocation examines two items by looking at two paths from root to leaf (only the leaves of the active 2-3 tree contain items). In any sequence of operations on a structure the paths chosen by `Hscan` or `STscan` advance through the tree in a standard "left to right" order. The path chosen in an initial state is unpredictable, but after examining the rightmost path, the next invocation returns to the leftmost path, so that all nodes will be examined in any sequence of (sufficiently many) operations.

For the composite data structure, we add a step to `Hscan` in only one occasion, which is immediately after the rightmost path of the heap is examined: the assignment $\texttt{Hbit} \leftarrow f_{\texttt{H}}(\texttt{Hbit}, \texttt{STbit})$ is executed, for

$$f_{\texttt{H}}(x,y) \;=\; \begin{cases} x & \text{if} \quad x = y \\ 1 - x & \text{if} \quad x \neq y \end{cases}$$

If $\texttt{Hbit} = 1 \;\wedge\; \texttt{STbit} = 1$ as a result of this assignment, then the assignment $\texttt{curcolor} \leftarrow \texttt{nextcolor}$ is executed.

One change to `STscan` resembles the change to `Hscan`: immediately after `STscan` examines the rightmost path of the 2-3 tree, the assignment $\texttt{STbit} \leftarrow f_{\texttt{ST}}(\texttt{Hbit}, \texttt{STbit})$ is executed, where

$$f_{\texttt{ST}}(x,y) \;=\; \begin{cases} y & \text{if} \quad x \neq y \\ 1 - y & \text{if} \quad x = y \end{cases}$$

The remaining change to STscan adds extra work in the examination of a 2-3 tree item. When STscan examines an item (k, d), the procedure crosschecks (k, d); if the crosscheck indicates that k does not have a corresponding heap value, then STscan removes (k, d) from the active 2-3 tree. If the crosscheck test passes, STscan then evaluates Hinheap(d), and removes (k, d) from the active 2-3 tree if Hinheap(d) is *false*. Finally, if (k, d) passes crosscheck and Hinheap tests, then STscan assigns $c \leftarrow$ curcolor where c is the color variable of the heap item associated with k.

5.3 New Background Activities

Each invocation of a heap operation contains a call to Hscan. Each invocation of a 2-3 tree operation contains a call to STscan. The modifications of these background activities described in Section 5.2 supply most of the effort needed to stabilize the composite structure; only one additional, new procedure is needed. We call this new routine trimheap. Procedure trimheap is called once in the execution of any of the composite operations, and consists of steps shown in Figure 1. By reasoning about the running times of Hpointer, *crosscheck*, STintree, Hinheap, and Hdelete, it follows that any call to trimheap requires at most $O(\lg K)$ time at an arbitrary state and $O(\lg m)$ time at a legitimate state for a heap-search tree containing m items (at a legitimate state, the heap's root has an accurate *height* field, which then is used to limit the running time of Hinheap).

We also suppose that each of the composite operations include calls to STscan and Hscan. Such calls will already be included whenever an operation invokes one of the appropriate constituent operations, however not all composite operations invoke constituent operations on both heap and 2-3 tree structures. The find operation, for instance, invokes STfind but does not include any heap operation; and although deletemin invokes both Hdeletemin and STdelete in a legitimate history, it may not do so before the stabilization point, as we show in Section 5.4. Therefore we suppose that trimheap includes calls to Hscan and STscan as needed, to ensure these constituent background activities execute with each operation in any history.

5.4 Operations of the Composite

The four operations of the heap-search tree have psuedo-code listed in Figure 1. We suppose in this figure that if STfind(k) returns an item (a, b), then a pointer to item (a, b) in the 2-3 tree component is available or can be obtained in $O(1)$ time. Also, for the sake of brevity, we do not provide details for how to deal with duplicate key values; we assume that a STdelete(k) will remove the item return by a STfind(k) immediately preceding, and suppose similar behavior for insertions.

The usage of *crosscheck* in the code for deletemin bends our earlier explanation of checking item pointers. This *crosscheck*(p) should check that the 2-3 tree item referred to by p is an item (k, d) such that d points to the root of the

```
trimheap
    $q \leftarrow$ pop( )
    if ($\neg$Hpointer($q$) $\vee$ $\neg$crosscheck($q$)) return
    $p \leftarrow$ search tree item for $q$
    if $\neg$STintree($p$) return
    $t \leftarrow$ height field of heap's root
    if Hinheap($q, t$) then Hdelete($q$)
```

```
find($k$)
  if (STfind($k$) = missing)
      return missing
  $(a, b) \leftarrow$ STfind($k$)   // $a = k$
  $p \leftarrow$ address of item $(a, b)$
  if (STcross($p$) $\wedge$ Hinheap($b$))
      $(v, e) \leftarrow$ heap item via pointer $b$
      return $(k, v)$
  STdelete($k$)   // delete invalid key
  return err
```

```
delete($k$)
  if (STfind($k$) = missing) return ack
  $(a, b) \leftarrow$ STfind($k$)   // $a = k$
  $p \leftarrow$ address of item $(a, b)$
  if (STcross($p$) $\wedge$ Hinheap($b$))
      Hdelete($b$)
      STdelete($k$)
      return ack
  STdelete($k$)   // delete invalid key
  return err
```

```
deletemin( )
  $t \leftarrow$ Hdeletemin( )
  if ($t$ = empty) return empty
  $(v, (p, c)) \leftarrow t$
  if (crosscheck($p$) $\wedge$ STintree($p$))
      $(k, d) \leftarrow$ 2-3 tree item via pointer $p$
      STdelete($k$)   // delete $(k, d)$
      return $(k, v)$
  return err
```

```
insert($k, v$)
  $d \leftarrow$ temporary value
  if (STinsert($k, d$) = full)
    return full
  // locate ST item just inserted
  $(a, b) \leftarrow$ STfind($k$)
  $t \leftarrow$ address of item $(a, b)$
  $e \leftarrow (t,$ curcolor$)$
  if (Hinsert($v, e$) = full)
    STdelete($k$)   // backout
    return full
  replace $b$ within 2-3 item by
    $b \leftarrow$ address of (v,e) in heap
  return ack
```

Fig. 1. `trimheap` procedure and composite data structure operations.

heap — since the heap item t returned by `Hdeletemin` was located at the root of the heap prior to being removed.

5.5 Verification

For a given point t in a history of operations on the heap-search tree, let $\mathtt{H}_t$ be the bag of items in the active heap and let $\mathtt{ST}_t$ be the bag of items in the active search tree. Let A_t denote the *active multiset* at point t, defined by $A_t = \{\ (k, v) \mid k \in \mathtt{ST}_t \ \wedge\ v \in \mathtt{H}_t \wedge (k, v)$ satisfy the crosscheck relation $\}$. The expressions $|\mathtt{H}_t|$ and $|\mathtt{ST}_t|$ the number of items in the (respective) active structures. A state of the composite data structure is *ST-legitimate* if the search tree component of the state is legitimate as defined in [9]. A state of the composite data structure is *H-legitimate* if the heap component of the state is legitimate as defined in [8]. A state is *ST/H-legitimate* if it is both ST-legitimate and H-legitimate. A state is *legitimate* if (*i*) it is ST/H-legitimate, (*ii*) for each item x of the active search tree, there exists an item y of the active heap such that x and y are related by the crosscheck relation, and (conversely) (*iii*) for each item a of the active heap, there exists an item b of the active search tree such that a and b are crosscheck related. The availability and closure properties, based on these definitions, have simpler proofs of correctness than the proof of convergence. To show convergence, we define a segment of a history to be a *color phase* if the curcolor value is the same at each point in the segment. Define *colorsafe* to be the weakest predicate closed under heap-search operations such that for any point s where *colorsafe* holds, the state of the data structure is ST/H-legitimate and $(\forall x : x \in A_s : x.\texttt{color} \neq \texttt{nextcolor})$.

Lemma 4. Let s be an ST/H-legitimate point in a history and let $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$. The color phase containing point s terminates at some point w after at most $15m_s$ operations and $\max(|\texttt{H}_w|, |\texttt{ST}_w|) \leq 16m_s$. Following any point s at which ST/H-legitimacy holds and $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$, there occurs a point t within $O(m)$ operations of any history such that the data structure at point t is *colorsafe* and $m_t = \max(|\texttt{H}_s|, |\texttt{ST}_s|) = O(m)$.

For any point s in a history of operations on the composite data structure, let $depth_s(j)$ for $j \in \texttt{H}_s$ be the number of pop operations required to obtain a pointer to j from the lossy stack ($depth_s(j) = \infty$ if there is no pointer to item j on the lossy stack). Let $\overline{\texttt{H}}_s$ denote the bag of active heap items that for which the crosscheck relation does not hold, that is, $\overline{\texttt{H}}_s$ contains those active heap items that do not correspond to any active multiset item. Define *stacksafe*(m) to be the weakest closed predicate such that any point s where *stacksafe*(m) holds, the state of the data structure is *colorsafe* and $(\forall j : j \in \overline{\texttt{H}}_s : depth_s(j) < 3m)$.

Lemma 5. Following any *colorsafe* point s with $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$, there occurs a point t within $O(m_s)$ operations of any history such that the data structure at point t is *stacksafe*(m_s) and $m_t = \max(|\texttt{H}_s|, |\texttt{ST}_s|) = O(m_s)$.

Theorem 1. Within $O(m_s)$ operations of any history the state of the composite data structure is legitimate, where $m_s = \max(|\texttt{H}_s|, |\texttt{ST}_s|)$ for the initial point s of the history.

6 Conclusion

There are likely simpler ways to construct a stabilizing available structure with the signatures of the composite presented in Section 4 (two search trees, an augmented tree, etc), however our aim was to investigate the composition of a data structure from given components. Although we present a construction, our results concerning the larger question are somewhat negative: not everything is achievable given constraints of (conservative) availability and stabilization (the possibility of conflict between availability and fault tolerance was observed long ago [1]).

The reader may wonder whether the case of sequential data structure operations, without distributed implementation or concurrency, merits any investigation: after all, the very title of [2] is "self-stabilization in spite of distributed control" (recent definitions of self-stabilization refer only to behavior [14] and care not whether the system is distributed or sequential). Here, the difficulties to overcome have to do with two aspects of availability during convergence, namely the integrity of operation responses and the time bounds of operations. Both aspects have practical motivation: users of data structures and system designers of forward error recovery value the integrity of operation responses; and having guaranteed time bounds on operations is useful for the design of responsive applications in synchronous environments. It could be interesting to reconsider our constraint on the time bound, perhaps allowing some polynomial extra time during convergence.

References

1. SB Davidson, H Garcia-Molina, D Skeen. Consistency in partitioned networks. *ACM Computing Surveys* 17(3):341-370, 1985.
2. EW Dijkstra. EWD391 Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective,* pages 41–46, Springer-Verlag, 1982. EWD391's original date is 1973.
3. S Dolev. *Self-Stabilization.* MIT Press, 2000.
4. S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.
5. A Fekete, D Gupta, V Luchangco, N Lynch, A Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220:113-156, 1999.
6. S Ghosh, A Gupta, T Herman, and SV Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC'96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.
7. T Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1–17, 2000.
8. T Herman, T Masuzawa. Available stabilizing heaps. *Information Processing Letters* 77:115-121, 2001.
9. T Herman, T Masuzawa. A stabilizing search tree with availability properties. *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS'01)*, pp. 398-405, March 2001.
10. JH Hoepman, M Papatriantafilou, P Tsigas. Self-stabilization of wait-free shared memory objects. In *Distributed Algorithms, 9th International Workshop Proceedings (WDAG'95)*, Springer-Verlag LNCS:972, pp. 273-287, 1995.
11. B Lampson. How to build a aighly available system using consensus. In *10th International Workshop on Distributed Algorithms (WDAG'96)*, Springer-Verlag LNCS 1151, pp. 1-17, 1996.
12. M Li, PMB Vitanyi. Optimality of wait-free atomic multiwriter variables. *Information Processing Letters*, 43:107-112, 1992.
13. M Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9-22, 1990.
14. G Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.