Available and Stabilizing 2-3 Trees

Ted Herman[*]
University of Iowa
herman@cs.uiowa.edu

Toshimitsu Masuzawa[†]
Graduate School of Engineering Science
Osaka University
1-3 Machikaneyama, Toyonaka 560-8531, Japan
masuzawa@ics.es.osaka-u.ac.jp

1 December 2000

Abstract

Transient faults corrupt the content and organization of data structures. A recovery technique dealing with such faults is stabilization, which guarantees, following some number of operations on the data structure, that content of the data structure is legitimate. Another notion of fault tolerance is availability, which is the property that operations continue to be applied during the period of recovery after a fault, and successful updates are not lost while the data structure stabilizes to a legitimate state. The available, stabilizing 2-3 tree supports `find`, `insert`, and `delete` operations, each with $O(\lg n)$ complexity when the tree's state is legitimate and contains n items. For an illegitimate state, these operations have $O(\lg K)$ complexity where K is the maximum capacity of the tree. Within $O(t)$ operations, the state of the tree is guaranteed to be legitimate, where t is the number of nodes accessible via some path from the tree's root at the initial state. This paper resolves, for the first time, issues of dynamic allocation and pointer organization in a stabilizing data structure.

Keywords: data structures, fault tolerance, self-stabilization

1 Introduction

Two important themes in the literature of fault tolerant design are availability and self-repair. A "highly available" system continues to provide service (perhaps at degraded level) in spite of failures of its components. If component failures are transient, then the system can repair the states of damaged components. These themes also apply to abstract data structures,

[*]This work is supported by NSF CAREER award CCR-9733541.

[†]Supported in part by Japan Society for the Promotion of Science (JSPS), Grants-in-Aid for Scientific Research.

which is useful for object-oriented system design. We call a data structure available if each operation invocation returns a response consistent with its effect on the data structure, in spite of arbitrary values in all data structure fields (including pointers, keys, counters, and so on) prior to the operation. We call a data structure stabilizing if, for any initial state of the data structure, any sequence of operations applied to the data structure brings it to a legitimate state. Separately, availability and stabilization have drawbacks: availability does not guarantee repair to a damaged structure and performance of operations can remain permanently degraded; stabilization does not make guarantees about the behavior of operations in the period before repair has completed. We therefore seek data structures that are available and stabilizing.

(Self-) stabilization is the topic of numerous investigations in the field of distributed computing [9], but very few papers consider the question of stabilizing data structures. The usual model for stabilizing algorithms is process-oriented, meaning that variables subject to transient faults have dedicated processes that continually check and correct faulty data. We study passive data structures, for which the only checking and correction occurs within the normal application of operations (`find`, `insert`, `delete`); faults will not be corrected unless operations are applied.

Related to this work are papers such as [4] which consider transient corruption of one portion of data, but rely on control variables that initiate computation. Our assumption is that the data structure may be damaged, but each operation starts cleanly with its internal control variables uncorrupted. We wish to constrain the behavior of operations in cases where data is faulty by an availability guarantee, which resembles previous work on graceful degradation [5]. With the exception of a few recent papers [10, 3, 6], most stabilizing algorithms do not constrain behavior during periods while data is corrupt. Moreover, the stabilization time of our construction is adaptive: the stabilization time depends on the size of the initial (possibly damaged) tree structure, and in this respect our research follows a recent trend of adaptive stabilization times [1, 8].

Our contribution is a new form of stabilizing data structure, which constrains behavior of every operation, brings the the data structure to a legitimate state over a sequence of operations, and does so with an adaptive stabilization time. This paper goes beyond our previous investigation of heaps [7] by showing how availability and stabilization are possible for a dynamic data structure using pointers.

2 Stabilizing Search Tree Specification

The construction presented in this paper is one type of data structure supporting `find`, `insert`, and `delete` operations with logarithmic running times. The behavior of operations and the specification of stabilization properties are general, and we state them here in terms of a generic *search tree*. In this section, the behavior of a search tree is initially specified without considering stabilization properties. Subsequently this specification is revised to include stabilization criteria.

2.1 Search Tree Operations

A search tree is an associative memory containing *items* of the form $\langle key, datum \rangle$. The *capacity* K of the search tree is an upper bound on the number of items that the search tree may store. Let $\mathcal{H}$ be an infinite sequential history of operations on a search tree. Each operation consists of a pair $\langle inv, resp \rangle$ where the invocation *inv* is one of $\{$`find`,`insert`,`delete`$\}$ accompanied by calling parameters, and the response *resp* is as follows: for a `find` or `delete` invocation, the response is either "missing" or an item; for an `insert` invocation, the response is either "ack" or "full". The complete signature for an `insert` invocation is `insert`(*key*, *datum*), however to streamline the presentation we ignore the *datum* component and write `insert`(*key*) in subsequent sections. The signatures for the other invocations are `find`(*key*) and `delete`(*key*). We say an `insert` invocation *succeeds* if its response is "ack" and *fails* if its response is "full"; similarly, a `find` or `delete` invocation is said to fail if its result is "missing" and otherwise is successful.

Semantics of operations are given in terms of the content of the search tree, which we describe using the operation history. The search tree *content* is defined for any point between operations in a history $\mathcal{H}$. For completeness, we define the content before any operation in $\mathcal{H}$ to be the empty set (the search tree initially contains no items). Let t be a point in $\mathcal{H}$ between operations; the content of the search tree at point t, denoted by C_t, is the bag of items $C_t = I_t \setminus D_t$, where I_t is the bag of items successfully inserted prior to point t, and D_t is the bag of items successfully deleted prior to point t.

Search tree operations satisfy the following constraints: (*1*) a `delete`(k) (`find`(k)) invocation immediately following any point t in any history returns "missing" iff there exists no d such that $\langle k, d \rangle \in C_t$, and otherwise returns some item $\langle k, d \rangle \in C_t$; and (*2*) an `insert`$(k, d)$ operation immediately following any point t fails iff $|C_t| \geq K$, and otherwise returns "ack". From (*1*) and (*2*) one can show intuitive search tree properties, for instance, no `find` returns an item not previously inserted. A *balanced* search tree satisfies additional constraint, also specified with respect to any point t in a history: (*3*) the running time of any operation immediately following t is $O(\lg |C_t|)$.

2.2 Available and Stabilizing Operations

Transient faults inject arbitrary data into data structures, which is modeled in the literature of stabilizing algorithms by considering arbitrary initial states — the state following a transient fault is the "initial" state for subsequent computation. §2.1's characterization of search tree behavior depends on C_z being empty at the initial point z in any history, so we consider next a characterization admitting arbitrary initial content in a search tree.

Let $\mathcal{P}$ denote a history fragment, starting from an initially empty search tree, that consists entirely of successful `insert` operations. To specify behavior of $\mathcal{H}$ for an arbitrary initial search tree, let $\mathcal{H}' = \mathcal{P} \circ \mathcal{H}$. A search tree implementation is *available* if for every history $\mathcal{H}$ of operations there exists $\mathcal{P}$ such that $\mathcal{H}'$ satisfies (*1*) for all operations; and (*2'*) no `insert` operation at any point t succeeds if $|C_t| \geq K$. A balanced search tree implementation is available if it is an available search tree and the running time of every operation is $O(\lg K)$.

Note that (*2*) is not required for availability: an `insert` operation at a point t is allowed to fail when $C_t < K$. A trivial implementation of an available search tree would be one that fails all `insert` operations. Although this definition of availability weakens the specification, it does provide safety guarantees for the search tree content. For instance, if `insert`(k, d) does succeed, any subsequent `find`(k) will succeed at least until a `delete`(k) operation is applied.

Let $\mathcal{H}_v$ denote the suffix of a history $\mathcal{H}$ following a point v in $\mathcal{H}$. A search tree implementation is *stabilizing* if for every history $\mathcal{H}$ of operations there exists a point v and a history fragment P such that all operations in $\mathcal{P} \circ \mathcal{H}_v$ satisfy (*1*) and (*2*). A balanced search tree implementation is stabilizing if it is a stabilizing search tree implementation and every operation in $\mathcal{H}_v$ satisfies (*3*).

The point v in the definition of stabilization divides the history $\mathcal{H}$ into illegitimate and legitimate parts. Prior to v, the content of the search tree has no relation to the responses of invocations; the behavior could be chaotic in this portion of the history. Following v, all operations behave normally with respect to some "initializing" history $\mathcal{P}$. A possible implementation of a stabilizing search tree would be one that, after some number of operations, resets the content of the search tree to the empty set ($\mathcal{P}$ would be empty in this case). The portion of history $\mathcal{H}$ prior to point v is called the *convergence period*, and the worst case number of operations in the convergence period, taken over all possible histories for a stabilizing search tree implementation, is the *stabilization time* of the implementation.

Availability alone does not guarantee progress, since `insert` operations can continue to fail in a history although the search tree is not full. Stabilization does guarantee progress eventually, but items that have been inserted successfully during the convergence period could be lost before the search tree stabilizes. We therefore aim for a search tree that is both available and stabilizing. For example, a balanced, available, stabilizing, search tree enjoys safety and timing guarantees throughout any history (each operation has $O(\lg K)$ running time and the semantics of invocation responses are well-defined) and after convergence, behavior is what one expects of a balanced search tree.

3 Construction for Stabilization

Our stabilizing search tree is a modification of a conventional 2-3 tree implementation. After briefly reviewing this basic 2-3 tree, this section surveys the technical enhancements introduced for stabilization and availability.

3.1 2-3 Tree Review

A 2-3 tree is a balanced search tree with the following structure [2]: each non-leaf node has either two or three children and the path length from root to leaf is the same for every leaf. Each leaf contains one item, and non-leaf nodes contain one or two keys of items in their subtrees. Figure 1 presents an example 2-3 tree, showing how interior nodes have the maximum keys of their two left-most subtrees. It follows from this definition that a 2-3 tree of height h contains between 2^h and 3^h items.

This definition of a 2-3 tree differentiates between item nodes (leaves) and interior nodes, which is a detail not important for our presentation. In subsequent discussion the leaves of 2-3 trees are omitted; to show the keys of all items, parents of the omitted leaves list the key values of all their children. Figure 2 is an example with leaves omitted while all item keys are shown. Another interpretation of this representation is that two or three items are contained in each leaf of a tree of $n > 1$ items.

Operations on a 2-3 tree of n items (`find`, `delete`, `insert`) have $O(\lg n)$ running time because tree height is logarithmic. The `find` operation has a straightforward implementation; the

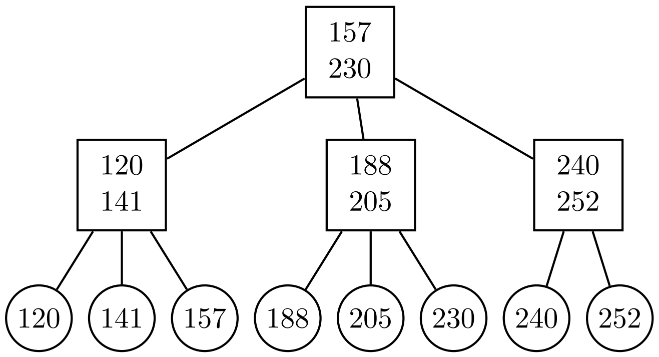

Figure 1: typical 2-3 tree

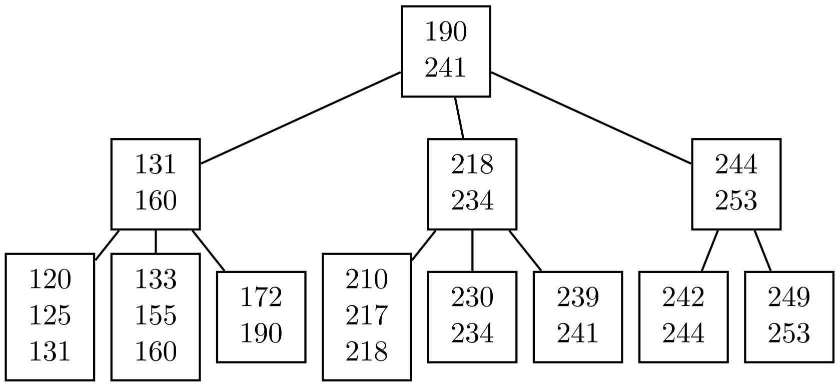

Figure 2: 2-3 tree with implicit leaves

`insert` and `delete` operations rearrange keys within interior nodes, possibly splitting nodes, merging nodes, or reassigning the root node location and adjusting tree height to maintain the invariant that each interior node have two or three children.

Our definition of availability depends on a maximum capacity K for a balanced search tree. Many texts describe tree operations in detail, but few consider the case of a 2-3 tree with fixed capacity. For simplicity of presentation, we suppose that K is a power of three, and fix the maximum path length from root to leaf at $pmax = \log_3 K$. A practical difficulty in defining capacity as a fixed threshold K for deciding success or failure of an `insert` operation is that a set of items can have numerous representations as a 2-3 tree, some of which make insertion harder than others.

For instance, no `insert` operation can succeed if it would increase the tree height beyond *pmax*, and requirement (*3*) for logarithmic running time precludes extensive key redistribution by any single `insert` operation. Consider the case of $pmax = 3$ (so $K = 27$) and the tree of Figure 2. A conventional implementation of `insert`(123) in this case would result in a recursive node split and increase in tree height; we make the design decision that `insert`(123) should fail here, although the tree has only 19 items. Formally, the consequence of this design decision is that definitions of 2-3 tree capacity and condition *(2)* for the success of an `insert` operation should be revised; to simplify the presentation we omit this level of detail.

The operation definitions in §2.1 refer to the content of a search tree as a bag of items, which allows for duplicate items in a search tree. Duplicate items in a 2-3 tree are usually differentiated by extending the key with a sequence number or a node address. We omit presenting details of this standard technique.

3.2 Structural Modifications for Stabilization

The literature of stabilizing algorithms is primarily oriented to distributed computing, in which a recurrent theme is establishing some global system property using processes with only local resources and limited communication facilities. Not surprisingly, the technique of many stabilizing algorithms is: processes detect illegitimate global states by local checking, and thereafter effect state correction either locally or by initiating a system reset. The problem of stabilizing 2-3 trees is not distributed, but shares some characteristics with distributed systems: the legitimacy of the data structure is a global property, but at most $O(\lg K)$ nodes can be visited by any single tree operation. We therefore follow some traditional stabilization techniques, starting with local checking to determine legitimacy of data. After describing some of the challenges posed by illegitimate data, we describe modifications to conventional 2-3 tree representations so that local checking is enabled.

We begin by considering how a transient fault can disrupt the content and organization of 2-3 tree. Such disruption has an impact on three entities, keys, pointers, and auxiliary variables. Key corruption violates the condition that each internal node key has the maximum item key value of the corresponding subtree. After a fault, key values can be duplicated and out of order. Pointer corruption damages the tree structure, possibly producing orphan nodes, ancestry cycles, and references to arbitrary locations in memory. Auxiliary variable corruption can cause the location of the root node to be lost, can invalidate counters, and damage the mechanism of node allocation and deallocation.

The standard implementation of a 2-3 tree uses $(k-1)$ keys in a node with k children. Our first modification is to give each node the same number of keys as it has children, and to strengthen navigation by using a pair of keys $(\texttt{low}_p, \texttt{high}_p)$ for each child p. The value $\texttt{low}_p$ is a lower bound on the minimum item value of the subtree rooted at p, and the value $\texttt{high}_p$

is an upper bound on the maximum key value of the subtree rooted at p. Thus each node has minimum and maximum bounds for all its children and their subtrees. This modification is a first step to local checking: it is now possible to verify key values by comparison between parent and child. The only keys remaining unverifiable are those of the items, which have no children. With this modification, each key pair within a node has a associated child pointer. If the pointer associated with a key is null, then that key is called *irrelevant*. Only relevant keys and their corresponding pointers are verified by local checking. We use the notation $\texttt{key}_p$ as shorthand for $(\texttt{low}_p, \texttt{high}_p)$.

Two modifications enable local detection of pointer corruption. The first is to support each tree link with a double pointer. Each node except the root now has a pointer to its parent. This provides a "sanity" check for pointers, so that a child pointer can be checked by comparing fields in two nodes: the pointer check, for relevant $\texttt{key}_q$ in node p, consists of testing the equality $\texttt{parent}(q) = p$. The situation of multiple parents of one node is now easily identified. However some global properties are not verified by this modification, including the property that every path from root to leaf should have the same length. The second modification addressing pointer corruption is to use a static allocation scheme for the placement of nodes in memory.

The storage used for allocation of tree nodes is partitioned into *pmax* segments labeled S_i, $0 < i \leq pmax$. Node placement in these segments invariantly satisfies: a node p at height i in the 2-3 tree resides in segment S_i (for completeness, we can let S_0 be the segment containing the data items in the tree). This constraint enables simple and local "type checking" of child and parent pointers: each child (parent) pointer of a node in segment S_i should refer to a node within segment $S_{(i-1)}$ ($S_{(i+1)}$). For a node p, let $\texttt{segment}(p)$ denote the segment containing p.

The partition of storage into segments $\{\, S_i \mid 0 < i \leq pmax\}$ dictates that allocation and deallocation of nodes occur within each segment. Each segment therefore has a free list of unallocated nodes, which is managed as a stack: a newly deallocated node is added to the front of the free list, and allocation consists of removing the first node in the free list. Invariantly, every "next" pointer of a node in free list of S_i should either be null or refer to some node within S_i. Each node in S_i should either be a tree node or an element of the free list. The total number of nodes in S_i is denoted $|S_i|$. For $i > 1$, $|S_i| = 1 + \lceil S_{i-1}/2 \rceil$, and $|S_1| = \lceil K/2 \rceil$.

Auxiliary to the segments, the following pointers are needed: a pointer `root` to the root node, and *pmax* additional pointers, which start the free lists of the segments. Figure 3 summarizes the structure of segments and auxiliary pointers as applied to the previous example of Figure 2. The symbol λ indicates a null pointer (which makes the corresponding key value irrelevant).

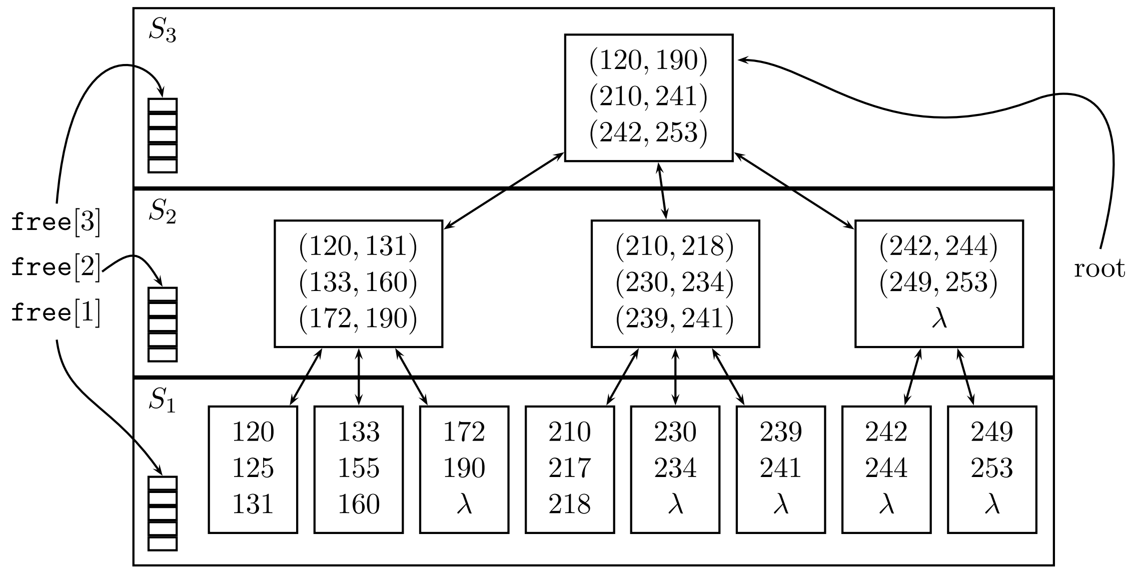

Figure 3: 2-3 tree within segments

4 Operation Modifications for Stabilization

Operations on a 2-3 tree are explained in many texts, and we suppose the reader is familiar with some conventional implementation of these operations, including node splits, merges, and recursive cases for these events. To simplify the presentation, we omit a full description of the operations and concentrate on the modifications needed for stabilization. For stabilization, `find`, `delete`, and `insert` operations use local checking to detect illegitimate conditions in the 2-3 tree and make corrections bringing the data structure to a legitimate state.

Two themes of the modifications to operations are truncation of the 2-3 tree to a legitimate fragment and background processes that reorganize the data structure. Our basic design decision is to trust key and pointer data from the tree's root downward, which has the consequence that an item in a 2-3 tree will be lost if a transient fault damages the path leading to that item. The rationale for this decision is due to the definition in the next subsection. We return in §6 to discuss further the issues of data loss due to transient faults.

4.1 The Active Tree

Given arbitrary initial (and possibly corrupt) values for variables and storage, many of the key and pointer properties of the 2-3 tree described in §3.2 could be falsified. Nevertheless, it is possible in many situations to identify some fragment, starting from the root, which enjoys properties of of a search tree.

A *state* of the 2-3 tree is a specification of the values for all variables and storage used by the data structure. The *active tree* is defined with respect to a given state σ. The definition of the active tree is recursive, depending on an intermediate tree called the *base tree*, denoted by T_σ. The items in the active tree define the data structure content; in proofs of availability and stabilization, the items of the initial active tree define the initial sequence $\mathcal{P}$ of successful `insert` operations.

Let T_σ be, for a state σ, the tree defined as follows. If $\texttt{root} = \lambda$ or $\texttt{segment}(\texttt{root}) \notin [1, pmax]$, then T_σ is empty; otherwise `root` specifies the root node of T_σ. The remaining nodes of T_σ are defined recursively: if $p \in T_\sigma$, and p has a relevant key $\texttt{key}_q$ such that $\texttt{parent}(q) = p$ and $\texttt{segment}(q) = \texttt{segment}(p) - 1$, then $q \in T_\sigma$.

The active tree is obtained by applying the following rules, as many times as possible, to the base tree (initially, let $T = T_\sigma$), giving priority to rule application in higher level segments over lower level segments, and giving priority to rules in the order listed where more than one rule is applicable to a particular node.

(a) if $p \in T$ has a key $(\texttt{low}_q, \texttt{high}_q)$ with $q \in T$ such that $\texttt{high}_q \leq \texttt{low}_q$, then remove q and its descendants from T.

(b) if $p \in T$ has a key $\texttt{key}_q$ with either $\texttt{segment}(q) = 1$ and q has no keys, or $\texttt{segment}(q) > 1$ and q has no relevant keys, then remove q and its descendants from T.

(c) if $p \in T$ has two relevant keys $(\texttt{low}_q, \texttt{high}_q)$ and $(\texttt{low}_r, \texttt{high}_r)$ that overlap ranges (such as $\texttt{high}_q > low_r$), then one of these relevant keys is made irrelevant by removing its associated child and all its descendants from T. The key to be made irrelevant is some deterministic choice; furthermore, if more than one key pair overlap ranges, the choice of which overlap to resolve is also deterministic (such determinism is necessary for a unique definition of the active tree).

(d) if $p \in T$ has a key $\texttt{key}_q$ for $q \notin T$, then remove q and all its descendants from T.

(e) if $p \in T$ has a relevant key $(\mathtt{low}_q, \mathtt{high}_q)$ and $q \in T$ has a relevant key $\mathtt{key}_r$ outside the range $(\mathtt{low}_q, \mathtt{high}_q)$, then remove r and its descendants from T.

The intuition of (a)–(e) is that keys at greater height in T_σ are more trustworthy than lower ones. So if a child has a key not reflected by its parent's key range for that child, some (or all) of the child's keys should be made irrelevant in the active tree.

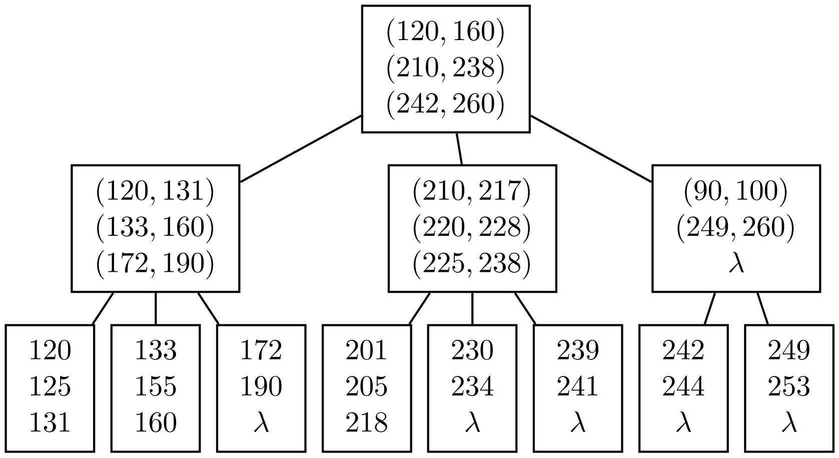

Figure 4: illegitimate keys in a 2-3 tree

Figure 4 shows an example of a 2-3 tree with illegitimate key values. The example has several violations of expected 2-3 tree properties: the root does not contain the maximum key values of all its children; the key value 228 is smaller than any item in the corresponding subtree; the key value 217 in S_2 does not equal the maximum key value of the corresponding child. After applying rules (a)–(e), the active tree pictured in Figure 5 results (assuming the key values at level S_1 of Figure 4 are legitimate).

4.2 Truncation

The structural modifications introduced in §3.2 enable each operation execution to perform extra measures of local checking and correcting without increasing the operation's time complexity. *Truncation* is one step in local correction: it assigns λ to selected pointers, bringing the tree closer to the definition of an active tree. *Node truncation*, for a node p, is the following procedure.

If $\mathtt{segment}(p) = 0$, there are no changes to p; otherwise,

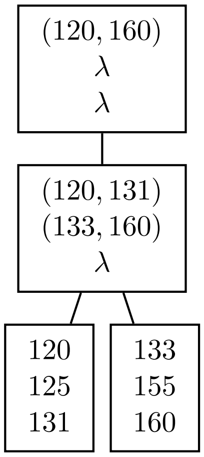

Figure 5: active tree of the illegitimate 2-3 tree

1. while a rule (a)–(d) is applicable to some relevant key $\texttt{key}_q$ in p, make $\texttt{key}_q$ irrelevant by assigning λ to the associated pointer for each q that is to be removed by (a)–(d);
2. for each relevant key $\texttt{key}_q$ in p, apply rule (e) as many times as needed to remove relevant keys from q; if q has no relevant keys as a result of this step, then apply rule (b) to make $\texttt{key}_q$ irrelevant by assigning λ to its associated pointer.

The node truncation procedure is applied in both preorder and postorder senses of visitation by each operation. That is, `find` (`insert`, `delete`) applies node truncation before examining or processing a node, in root to leaf order. The preorder application of truncation ensures that the operation does not follow paths outside the active tree. Postorder application of node truncation speeds up stabilization. Because truncations at lower levels can occur after the preorder processing of a given node, a postorder repetition of the node truncation procedure may result in additional changes bringing the tree to its active form. Note that the time overhead of node truncation is $O(1)$, so truncation does not increase the time complexity of operations.

4.3 Merge and Collapse

Truncation brings a tree with illegitimate keys and pointers to the form of an active tree. However as Figure 5 shows, active trees may not be 2-3 trees, because there may be single-child nodes. Two procedures convert such an unbalanced tree to a 2-3 tree: merging "only child" nodes with siblings and collapsing the root. The second modification for local checking and correction is to have `find`, `insert`, and `delete` apply a merge and collapse procedure after truncation.

The *merge and collapse* procedure deals with a node p (after applying the truncation procedure) that has a single child. There are three cases for p:

1. if p is the root with only child q, then assign `root` $\leftarrow q$ (this is a *collapse*);
2. if p is not the root, p has parent s, and p has a sibling r such that r contains at most two relevant keys, then merge nodes p and r and adjust keys and pointers of s appropriately;
3. if p is not the root, p has parent s, and p has a sibling r with three relevant keys, then move one key k (and its associated child) from r to p and adjust keys of s appropriately (the choice of k will be the largest or smallest key of r, depending on the sibling relation between p and r).

Cases 2 and 3 are not exclusive; in a situation where both cases exist, some choice (deterministic or nondeterministic) is acceptable to implement the procedure.

Each operation on the data structure, after applying truncation, then applies the merge and collapse procedure. Since the overhead for merge is $O(1)$, the time complexity of an operation does not increase due to the merge procedure.

4.4 Node Allocation

A successful `insert` operation increases the number of nodes in the tree and may increase the height as well. While conventional 2-3 implementations allocate and deallocate nodes, the modifications we introduce are to allocate on a segment basis and to locally check nodes on a free list to verify their availability.

We describe the allocation scheme in terms of an array of node structures for each segment. Auxiliary to each segment S_i, let `free`$[i]$ point to the head of the free list of unallocated nodes (as illustrated in Figure 3). We make the following convention: p is *detached* iff p is in the free list, `root` does not point to p, and `parent`$(p) = \lambda$. Node allocation can occur at S_i iff `free`$[i] = p$ where p is a detached node within segment S_i. Thus it is not sufficient for a node to appear on a free list for it to be detached — the local check verifies `parent`$(p) = \lambda$. The *size* of a free chain of S_i is defined to be the number of detached nodes, counting from S_i's free chain pointer, until either a (next) pointer leads outside of S_i or leads to a node that is not detached. The size of S_i's free chain is denoted by f_i.

A precondition for successful insertion is that sufficiently many nodes are detached, so an `insert` operation must test for detached nodes to determine whether the operation will succeed or fail. A simple implementation of such a test is to provide one extra node (beyond what is required for the capacity K) in each segment. An `insert` then fails if, at any level from the root down to S_1, there are no detached nodes. This test takes $O(\ell)$ time, where ℓ is the height of the tree.

Node deallocation occurs due to `delete` operations, merge, or collapse steps. The action for deallocation is a straightforward push onto the free list for the segment and assigning λ to the nodes's parent pointer.

4.5 Refusal

Operations succeed or fail in a legitimate 2-3 tree depending on the tree content, but conventional implementations do not encounter situations of single children, paths not terminating at items, and so forth. A node with an only child does not cause an operation to fail, since a postcondition of truncation is that the node's key is equal to some key of its child. However a path from the root, guided by key values, which does not end up at segment S_1, prevents operation completion (for instance, a path may not lead to a node in S_1 because truncation terminates the path). Operations fail in these cases.

Thus an `insert` operation fails, yielding a "full" response, if the insertion path prematurely ends — although sufficient free nodes for insertion may exist. Another instance of `insert` failure results when detached nodes in all segments do not exist, even if the number of items in the tree is far less than the tree's capacity.

4.6 Background Cleaning

Operations modified to include truncation, merge and collapse procedures can convert an illegitimate tree into a legitimate 2-3 tree, provided a sequence of these operations have an appropriately diverse set of key parameters. Of course, we cannot depend on the good fortune of operation parameters to stabilize the data structures. Moreover, the modifications described above do not address problems of illegitimate free lists. The remaining tasks of stabilization can be called "background cleaning" of the data structure. We describe these tasks first as concurrent activities, and later show how they can be integrated into the sequential operations on the data structure without increasing operation time complexity.

A straightforward approach to correcting an illegitimate tree would be a systematic visitation of all nodes reachable from the root, applying truncation and merger from lowest to highest segments. Such a systematic visitation could be a continual background activity. A complication with this approach is that operations may be applied to the data structure concurrently. To avoid this complication, we describe a visitation of the nodes that is easily interleaved with operations. Let `locate` be an internal operation differing from `find` only in its failure response: instead of returning "missing" if a key k is not contained in the tree, the response to `locate`(k) will be the smallest key k' such that $k' > k$ if such k' exists in some tree item, otherwise `locate`(k) returns "missing". Many implementations of search trees provide an operation similar to `locate` so that enumerations of the tree's items can be easily programmed. The implementation of `locate` applies truncation, merge and collapse steps as described above. The background activity consists of the continual repetition of the following: invoke `locate`(t), where t is the "current" key value; if the response is "missing", then assign to t the least possible value in the domain of key values, otherwise assign to t the next greater possible value than the key value in the response. The background activity is thus a round-robin visitation of the items of the tree. Observe that the initial current key value t is unimportant to this activity.

The remaining issue for background activity is the collection of orphan nodes that should belong to free lists. There are *pmax* such background activities, one for each segment. For segment S_i this activity consists of a scan of all nodes within S_i in round-robin order. The scan tests each node p with an `intree`(p) function to determine whether or not p is contained in the tree; if p is not in the active tree, then p is pushed onto the free list for segment S_i.

The `intree`(p) test has a recursive definition. If p is the root, then `intree`(p) is defined to be *true*. Cases where `intree`(p) is *false* are: if `parent`(p) does not have a corresponding relevant key and child pointer to p; if `segment`$(p) \geq$ `segment`(`root`) for $p \neq$ `root`; or if `segment`(`parent`(p)) $\neq$ `segment`$(p) + 1$. Finally, if none of these cases apply, then the definition is recursive: `intree`(p) = `intree`(`parent`(p)). The worst-case running time for `intree`(p) is proportional to the height of the root. In an arbitrary initial state of the data structure, `intree`$(p) =$ *true* does not imply that p is part of the active tree, however, `intree`$(p) =$ *false* does imply that p not in the active tree. After the data structure stabilizes to a legitimate 2-3 tree by sufficiently many truncation and merge steps, any subsequent `intree`$(p) =$ *true* does imply that p is in the active tree.

The `intree` test identifies orphan nodes by a negative response, but the negative response is also returned for nodes in the free list. Thus `intree` does not precisely identify those nodes that should be in the free list but are not currently in the free list. Our approach is to force any p for which `intree`(p) is *false* to have detached status (i.e., assigning `parent`(p) $\leftarrow \lambda$) and then moving it to the front of the free list for its segment. So that the number of nodes in a free list are not decreased, the free list has a bidirectional pointer implementation: moving any node in the free list to the front takes $O(1)$ time and the number of nodes in the free list is unchanged.

4.7 Cleaning in Operations

The 2-3 tree is a passive data structure with no independent, autonomous processes to perform background cleaning activities. Therefore, each operation on the data structure contributes some processing to the background activities. Another way to state this is that a sequence of data structure operations simulates background processes in addition to normal work of the operations. Seen as an ongoing simulation, the various background processes require some state information that is saved when the simulation is suspended between operations, and restored at the start of each operation to continue the simulation. The state information for such suspended background activity results in the following auxiliary variables: `curkey` contains the key value used in the `locate` traversal of the tree's nodes; `curnode`[i], for $0 < i \leq pmax$, is the current node location in segment S_i for the round-robin collection of free nodes; and `count` is an integer counter used to control the rate of the simulation.

Each data structure operation (`insert`, `delete`, `find`) contributes to cleaning by invoking `locate` twice and attempting eleven free node collections (that is, subjecting eleven nodes to the `intree` test and moving nodes not in the tree to the free list). Each `locate` uses and increments `curkey`, and each collection attempt advances the round-robin `curnode` location. Not all of the eleven collection attempts occur in the same segment, nor does each operation repeat the same selection for the collection attempts: the value of `count` determines, for each attempt, which segment is chosen. For each attempt, the segment choice is S_i where i is the largest positive value such that `count` mod $2^{i-1} = 0$, with `count` $\leftarrow$ (`count` $+ 1$) mod K executed after each collection attempt.

Lemma 1 For any sequence of k data structure operations, for $0 < i \leq pmax$, at least $\lfloor 11k/2^i \rfloor$ collection attempts occur in segment S_i.

Proof: The k data structure operations generate $11k$ collection attempts, and for each attempt finding `count` odd, segment S_1 is chosen. Thus either $\lfloor 11k/2 \rfloor$ or $\lceil 11k/2 \rceil$ collection attempts occur at S_1, and the remaining collection attempts occur in some S_i, $1 < i \leq pmax$. A similar observation holds for S_2, and recursively, to verify the lemma. $\square$

5 Verification

Theorem 1 The construction of §4 satisfies availability.

Proof: The proof is a straightforward verification that all operations (`insert`, `delete`, `find`) consult and modify only the active tree, and background activities do not remove items from the active tree. Thus the content of the search tree is defined as the set of items contained in the S_1 nodes of the active tree. $\square$

Notation. Let $\bar{n}$ be the number of items in the initial active tree and let $\bar{m}$ be the number of nodes in the initial base tree. Let $\bar{n}_i$ be the number of active tree nodes in segment S_i; we also denote $\bar{n}$ by $\bar{n}_0$. If the initial active tree is a 2-3 tree, it follows that $\bar{n}_i \leq \bar{n}_{i-1}/2$, so $\bar{n}_i \leq \bar{n}/2^{i-1}$. The overbar notation refers to the initial active tree, and we remove the overbar when counting nodes at subsequent points in a history of operations. Thus n_i is the number of active tree nodes in S_i at a specified state, $\bar{f}_i$ is the initial size of the free chain in S_i, and f_i is the size of the free chain at a specified state. To refine the node counts, let n_i^r be the number of active tree nodes with r children in segment i, $i > 1$, and be the number of tree nodes with r relevant keys for $i = 1$. Thus $\bar{n}_2^3$ is the number of active tree nodes with three children in S_2 at the initial state.

Call any state σ a *normal state* if the base tree T_σ equals the active tree of σ and this active tree is a 2-3. A state σ is *safe* if it is a normal state and for every segment S_i, either $f_i + n_i = |S_i|$ or $f_i \geq 2n_i$.

Lemma 2 Any data structure operation applied to a normal state results in a normal state.

Lemma 3 Starting from any initial state of the data structure, the active tree equals the base tree, and the active tree is a 2-3 tree with $n = O(\bar{m})$, within $O(\bar{m})$ operations of any history.

Proof: The proof relies on arguments using truncation and background cleaning to show that $O(\bar{m})$ operations are sufficient to visit, check, and correct all nodes of the initial base tree (including the possibility that successful `insert` operations increase the size of the base tree during this sequence of $O(\bar{m})$ operations). □

Texts describing 2-3 trees or B-trees observe that the frequency of node splits and merges decreases geometrically with tree height. Such observations are simple to verify given an initially empty tree and then considering worst-case sequences of operations. For our purposes, this observation should be enhanced to consider an initially non-empty tree.

Lemma 4 In a sequence of t operations, $t > \bar{n}_i$, at most $\bar{n}_i + \lceil (t - \bar{n}_i)/2^i \rceil$ node splits occur in segment S_i.

Proof: In the worst case, each of the $\bar{n}_i$ tree nodes in S_i have three children each, so $\bar{n}_i$ of the t operations can split these initially present nodes. The usual maximum rate of splitting is once per 2^i `insert` operations, and the lemma states both of these observations. □

Lemma 5 Any sequence of $\bar{n}$ operations applied to an initially safe state results in a safe state, and no `insert` operation fails during this sequence of operations unless $n > K$.

Proof: To reason about progress over the course of a sequence of operations on the data structure, a type of variant function is useful. We use for each segment S_i a four tuple $\langle n_i^3, \quad n_i^2, \quad f_i, \quad c_i \rangle$, where n_i^3, n_i^2, and f_i are defined above, and c_i is the number of collection attempts that have previously occurred in S_i (from the initial state to the current state). The evolution of this tuple for different types of operations is summarized as follows.

$$\begin{array}{lcl}
\langle n_i^3, \; n_i^2, \; f_i, \; c_i \rangle & \xrightarrow{(a)} & \langle n_i^3 - 1, \; n_i^2 + 2, \; f_i - 1, \; c_i + \delta_i \rangle \\
\langle n_i^3, \; n_i^2, \; f_i, \; c_i \rangle & \xrightarrow{(b)} & \langle n_i^3 + 1, \; n_i^2 - 1, \; f_i, \; c_i + \delta_i \rangle \\
\langle n_i^3, \; n_i^2, \; f_i, \; c_i \rangle & \xrightarrow{(c)} & \langle n_i^3, \; n_i^2, \; f_i, \; c_i + \delta_i \rangle \\
\langle n_i^3, \; n_i^2, \; f_i, \; c_i \rangle & \xrightarrow{(d)} & \langle n_i^3 + 1, \; n_i^2 - 2, \; f_i + 1, \; c_i + \delta_i \rangle \\
\langle n_i^3, \; n_i^2, \; f_i, \; c_i \rangle & \xrightarrow{(e)} & \langle n_i^3 - 1, \; n_i^2 + 1, \; f_i, \; c_i + \delta_i \rangle
\end{array}$$

In this table, δ_i represents a nondeterministic number of collection attempts in S_i (ranging between zero and five) addressed by Lemma 1 for a single operation. The types of operations in the table are (a) a node split, (b) a key insertion without a split, (c) a `find`, unsuccessful `insert` or `delete`, or a successful `insert` or `delete` affecting segments below S_i but making no change at S_i, (d) a node merge, and (e) removal of a key without a merge. Only the transition (a) increases the number of tree nodes in a segment, and we introduce simpler notation for this, summing n_i^3 and n_i^2 to make a triple:

$$\langle n_i\,,\quad f_i\,,\quad c_i\rangle \xrightarrow{(a)} \langle n_i+1\,,\quad f_i-1\,,\quad c_i+\delta_i\rangle$$

Although the additive factor δ_i in the table above is indeterminate, Lemma 1 does provide a lower bound for a sequence of operations. Since our goal is to establish sufficient free chain size, we consider the worst case sequence of operation to deplete a free chain, namely a sequence of type (a) transitions. For a sequence of t data structure operations starting from the initial state, with $t > \bar{n}$, the result of the sequence satisfies

$$\langle \bar{n}_i\,,\quad f_i\,,\quad 0\rangle \xrightarrow{r_i\times(a)} \langle \bar{n}_i+r_i\,,\quad f_i-r_i\,,\quad s_i\rangle$$

where r_i, the number of type (a) transitions in segment S_i satisfies $r_i \le \bar{n}_i + \lceil (t-\bar{n}_i)/2^i \rceil$ by Lemma 4, and $s_i \ge \lfloor 11t/2^i \rfloor$ by Lemma 1. Because we require bounds only we can write $r_i \approx (t+\bar{n}_i)/2^i$ and $s_i \approx 11t/2^i$. Approximate bounds are expressed by

$$\langle \bar{n}_i\,,\quad f_i\,,\quad 0\rangle \xrightarrow{r_i\times(a)} \langle \bar{n}_i+(t+\bar{n}_i)/2^i\,,\quad f_i-(t+\bar{n}_i)/2^i\,,\quad 11t/2^i\rangle$$

Also, the initial value $\bar{n}_i$ lies in the range $[\bar{n}/3^i, \bar{n}/2^i]$ by the definition of a 2-3 tree, so a conservative bound on the free chain size is given by

$$\langle \bar{n}/2^i\,,\quad f_i\,,\quad 0\rangle \xrightarrow{r_i\times(a)} \langle \bar{n}/2^i+(t+\bar{n}/2^i)/2^i\,,\quad f_i-(t+\bar{n}/2^i)/2^i\,,\quad 11t/2^i\rangle$$

We now distinguish between two cases, $\bar{n} = O(K)$ and $\bar{n} = o(K)$. For the case $\bar{n} = O(K)$ recall that each segment S_i has approximately half of the elements of S_{i-1}, with S_1 having about $K/2$ elements (so that, if each element is a node with two items, the capacity K has been attained). It follows that within $O(\bar{n})$ operations, every element of every segment undergoes a collection attempt. Thereafter, each element of S_i is either on the free chain for S_i or is a node in the active tree. In such a state, an `insert` operation fails only if the active tree contains at least K items, which establishes the theorem's conclusion.

For the case $\bar{n} = o(K)$, we examine a history of $\bar{n}$ operations ($t = \bar{n}$). For bounding the free chain size, we then have

$$\langle \bar{n}/2^i\,,\quad f_i\,,\quad 0\rangle \xrightarrow{r_i\times(a)} \langle \bar{n}/2^i+(\bar{n}+\bar{n}/2^i)/2^i\,,\quad f_i-(\bar{n}+\bar{n}/2^i)/2^i\,,\quad 11\bar{n}/2^i\rangle$$

An overestimate of the count of nodes and size of free chain is obtained by the substitution of $\bar{n}$ for $\bar{n}/2^i$, given by

$$\langle \bar{n}/2^i\,,\quad f_i\,,\quad 0\rangle \xrightarrow{r_i\times(a)} \langle 3\bar{n}/2^i\,,\quad f_i-2\bar{n}/2^i\,,\quad 11\bar{n}/2^i\rangle$$

Thus we see $11\bar{n}/2^i$ collection attempts in S_i exceeds the number of active tree nodes by at least $8\bar{n}/2^i$; this implies that after the $\bar{n}$ operations, at least $8\bar{n}/2^i$ collection attempts occur outside of active tree nodes. Of course, some or all of these collection attempts could apply to elements already in the free chain. So, while not every collection attempt outside the active tree results in an increase in the free chain size, the $11\bar{n}/2^i$ collection attempts do ensure a free chain size of at least $8\bar{n}/2^i$, less any elements consumed by splits during the period of these collection attempts. Since the number of elements consumed is $2\bar{n}/2^i$ during this period, it follows that the free chain size is at least $6\bar{n}/2^i$. Thus if $f_i = 2\bar{n}/2^i$ (the minimum needed to permit all the splits), then the size of the free chain after $\bar{n}$ operations is at least $6\bar{n}/2^i$ in segment S_i.

A conclusion of this analysis is that $\bar{n}$ operations at most triple the number of tree nodes in S_i, while multiplying the free list size by a factor of six. The analysis also shows that $f_i \geq 2\bar{n}_i$ is sufficient to supply all node allocation. Hence, the result of applying $\bar{n}$ operations supplies sufficiently many free list elements for a subsequent sequence of $\bar{n}$ operations (because $6\bar{n}/2^i$ is twice $3\bar{n}/2^i$). □

Lemma 6 Any sequence of $\bar{n}$ operations applied to an initially normal state results in a safe state.

Proof: The analysis presented in the proof of Lemma 5 holds for purposes of bounding the free chain size even when operations are not of type (a), which shows that after $\bar{n}$ operations every free chain size either includes all non-tree nodes or is double the number of free chain nodes. □

Theorem 2 The construction of §4 is stabilizing with $O(\bar{m})$ stabilization time.

Proof: Lemmas 2 and 5 show stability for a tree in safe state. Lemma 3 states that $O(\bar{m})$ operations suffice to reach a normal state, and Lemma 6 implies that within $O(\bar{m})$ subsequent operations, the state is safe. □

6 Discussion

The construction presented here shows that goals of availability and stabilization are achievable for a search tree. The solution is adaptive, however the adaptive stabilization period is related to the number of nodes of the initial active tree rather than the number of items (in [7] the adaptivity is linear in the size of the initial active heap). This seems unavoidable, since the active tree could initially have only one item, but $pmax = O(\lg K)$ nodes, and it is not possible to recognize the active tree and truncate it by $O(1)$ operations that are each limited to $O(\lg K)$ running time.

An important issue not addressed in this paper is limiting the amount of data lost due to a transient fault to be proportional to the scope of that fault. If the root of the tree is lost, then all of the items of the data structure are lost by our construction. So, in the worst case, damage to a single node can lead to loss of all data. At the other extreme, damage to a leaf node only results in loss of the data at the leaf. If minimizing data loss is an important goal, then data could be secured in higher level segments of the tree using replication techniques [11, 13, 12] to reduce the probability of loss by a transient fault. The degree of replication could be made proportional to the height of the node. If fault probability distributions have location independence, then it could be that the probability of losing data is roughly uniform for any node (least likely at higher levels due to replication, but with larger impact when it does occur). Using such a replication would have added storage cost and also a cost in operation times, since each operation would verify sufficient consistency among replicas.

References

[1] Y Afek and S Dolev. Local stabilizer. In *Proceedings of the 5th Israeli Symposium on Theory of Computing and Systems*, pp. 74-84, 1997.

[2] AV Aho, JE Hopcroft, JD Ullman. *The Design and Analysis of Computer Algorithms*, 1974, Addison-Wesley.

[3] S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.

[4] MJ Fischer, S Moran, S Rudich, and G Taubenfeld. The wakeup problem. *SIAM Journal on Computing*, 25:1332-1357, 1996.

[5] M Herlihy, JM Wing. Specifying graceful degradation in distributed systems. *ACM TODS*, 19(4):586-625(1994).

[6] T Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1-17, 2000.

[7] T Herman, T Masuzawa. Available stabilizing heaps. To appear in *Information Processing Letters*, 2000.

[8] S Kutten, B Patt-Shamir. Time-adaptive self stabilization. In *Proceedings of the 16th Annual ACM Symposium on the Principles of Distributed Computing*, pp. 149-158, 1997.

[9] M Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45-67, 1993.

[10] E Ueda, Y Katayama, T Masuzawa, and H Fujiwara. A latency-optimal superstabilizing mutual exclusion protocol. In *WSS'97 Proceedings of the Third Workshop on Self-Stabilizing Systems*, pp. 110-124, 1997.

[11] MO Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM* 38:335-348, 1989.

[12] Y Aumann, MA Bender. Fault tolerant data structures. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pp. 580-589, 1996.

[13] S Kutten, D Peleg. Fault-local mending. In *Proceedings of the 14th Annual ACM Symposium on the Principles of Distributed Computing*, pp. 20-27, 1995.