Recoverable Concurrent Data Structures: A Methodology Approach

Anonymous Author(s)

Abstract

Recent developments foreshadow the emergence of new systems, in which byte-addressable *non-volatile main memory* (*NVRAM*), combining the performance benefits of conventional main memory with the durability of secondary storage, co-exists with (or eventually even replaces) traditional volatile memory. Consequently, there is increased interest in *recoverable* concurrent objects: objects that are made robust to crash-failures by allowing their operations to recover from such failures. This paper presents a principled approach to deriving recoverable versions of several widely-used concurrent data structures. Specifically, the approach can be applied in a wide range of well-known concurrent data structure implementations to make them recoverable, including stacks, queues, linked lists, trees and elimination stacks.

Keywords keyword1, keyword2, keyword3

1 Introduction

Recent years has seen the emergence of systems in which byte-addressable *non-volatile main memory* (*NVRAM*), combining the performance benefits of conventional main memory with the durability of secondary storage, co-exists with (or eventually even replaces) traditional volatile memory. This has lead to increased interest in the *crash-recovery* model, in which a failed process may be resurrected after it crashes. Of particular interest is the design of *recoverable concurrent objects* (also called *persistent* [6, 7] or *durable* [19]): objects that are made robust to crash-failures by allowing their operations to recover after such failures.

It is challenging to design data structures that persist in the presence of crashes and recoveries, and several concurrent implementations were proposed. While many of these exploit specific aspects of an object to optimize the implementation, it is important to develop generic approaches for deriving recoverable implementations from their non-recoverable counterparts. Such an approach should preserve the structure and efficiency of the implementation, as much as possible, while avoiding the need to design new algorithms.

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper presents a principled approach to deriving recoverable objects and describe it in detail through three widely-used concurrent data structures: a linked list [12], binary search tree [] and elimination stack [13].

Our results are presented in the context of an abstract individual-process crash-recovery model for non-volatile memory [3], in which processes communicate via non-volatile shared-memory variables. At any point, a process may incur a crash-failure, causing all its local variables, [[[except for its program counter,???]]] to be reset to arbitrary values. Operation response values are returned via volatile processor registers, which may become inaccessible to the calling process if it fails just before persisting the response value. [[[Each data structure operation???]]] has an associated *recovery function* that is responsible for restoring it upon the recovery from a crash-failure. The recovery function completes the current outstanding operation on the data structure, if there was any, returning either its *response* or a *fail* indication, if it was unsuccessful. Both responses are consistent with the resulting state of the data structure, to which the operation was applied (in the former case) or not (in the latter case).

Related Work

Several different correctness conditions and implementations have been proposed for recoverable data structures [1, 5–7, 10, 11, 16, 19]. However, this work concentrates on maintaining the consistency of the concurrent object in the face of crash failures, and does not consider *detectability*, that is, the ability to conclude upon recovery whether the crashed operation took effect.

There is a universal implementation [8] which is both durable [[undefined?]] and lock-free, it uses only read, write and CAS. This implementation applies at most one *persistence[[?]] fence* (flushing the contents of the memory) per operation, which is optimal, but it keeps the entire history of the object in a designated shared queue; it also keeps a per-process persistent log, such that these logs together keep the entire history, and different logs may have a big overlap. Furthermore, to determine the response of an operation, it is necessary to read the entire history up to its linearization point. This construction can easily be made detectable since an operation was linearized if and only if it appears in the shared queue (representing the object's history). Specifically, after the system completes its recovery routine, in the recovery from an $Update(op)$, the process can determine the response value from the shared queue; this assumes that each *op* is uniquely identified. This work considers a *system-wide* crash model, in which all processes crash together and a single recovery function is executed upon recovery, in order to consistently reconstruct the queue data structure. It ensures *durable linearizability* [19]: after a full system crash, the state of the object must reïňĆect a consistent operation subhistory that includes all operations completed by the time of the crash, i.e., crashed operations may get lost.

Nesting-safe recoverable linearizability (*NRL*) [3] is a novel crash-recovery model together with a correctness condition. It associates each recoverable operation with a recovery function, invoked after a crash in the operation to help a process to complete the operation, as well as restore the response value if needed. They give recoverable implementations for read, write and CAS. As suggested by its name, NRL support *nesting*, so taking an algorithm that uses only read, write and CAS, and replacing each primitive with its recoverable version yields an NRL implementation. Some minor changes are still needed in order to use this transformation, but they hold for all the implementations presented in that paper. However, this transformation is quite costly, in terms of both time and space.

Indeed, implementations using only read, write and CAS can be made recoverable and detectable [4], by partitioning the code into *capsules*, each containing a single CAS followed by some number of reads, and replacing each CAS with its recoverable version. This allows to recover from a crash inside the capsule. *Normalized* implementations [18] can be further optimized so that an operation contains only two capsules. However, not all implementation are given in a normalized form, and converting an implementation into a normalized form may be costly by itself. This general transformation has several drawbacks: For example, replacing a CAS with its recoverable variant requires each CAS to have distinct arguments, ensured by adding unique sequence numbers, which are stored in the CAS location. This means that CAS is applied to words with unbounded length, even if the original implementation applied CAS over a finite domain. Furthermore, although two capsules are used for each normalized operation, these two capsules are repeatedly executed in attempt to complete the operation. Thus, the implementation is lock-free.

In many cases we can avoid it, without the extra capsule complexity for a failed attempt. Moreover, we would like the transformation to change as little as necessary from the original code, even at the price of having a costly recover function. Assuming crashes are rare, this may yield a more efficient implementation in practice.

2 Overview of Our Approach

Our methodology separately considers the operations provided by a linearizable implementation A of a data structure Q, according to their properties. In this initial description, we assume each operation becomes visible and is linearized through at most a single *CAS*; this simplifies recovery by avoiding scenario of an operation changing the data structure but is yet to complete and be linearized. (Later in this section, discuss how to remove this assumption, in the context a binary search tree.)

Consider an implementation of an operation *Op* that does not change the data structure, as is often the case with a search operation. In this case, the operation remains intact, and the recovery function simply restarts it from scratch. (Later, in the context of the linked list, we explain how this approach is extended to the case where stopping *Op* at any point and restarting it does not violate linearizability.)

For operations that do change the data structure, assume first that there is a way to uniquely identify each instance of *Op*. In this case, *Op* can be made recoverable by adding an indication once it is visible. The recovery function then checks for this indication: if found, the operation has completed, and its response is returned; otherwise, the operation is restarted. For example, assume process p tries to add a new unique node *nd* to the data structure using *CAS* (other processes may add different nodes with the same data). Once *nd* is added, it can be found in the data structure, indicating that the operation has completed. However, this indication is lost when *nd* is removed. This problem can be overcome by flagging a special field in *nd* before removing it. This way, we have two indicators, one is set once *nd* is added and the other when *nd* is deleted.

A further complication is posed when the implementation employs a helping mechanism, namely, several processes attempt to perform an operation, which is not uniquely identified. When p crashes during an execution of *Op* and recovers, it might be able to check whether *Op* is complete, but it is unable to know whether it is the one to perform it, or some other process. As a result, p may not be able to recover the right response value. This is resolved by adding an *owner* field (for each operation type), used for agreeing which process performed the operation; after a process p completes an operation, *CAS* its id to *owner*, to declare itself as the winner. If p crashes during an execution of *Op* and recovers, p first checks whether the operation is visible, and if so, p tries to set *owner* to p with *CAS* and then responds according to the id stored in *owner*. (This is demonstrated in the *delete* of our Linked-List implementation.)

In all cases, it is necessary to persist the response value before returning it. In some cases, p may also need to store some recovery data; for example, the node to be added or removed. This information is stored in a designate memory location, for each process; after the process stores this information and before the process starts performing the operation, a checkpoint is set to signify that the data is relevant for the current operation.

We next further explain our approach with a relatively simple example, of a linked-list set, and then explain some extensions with a *binary search tree*. Later in the paper (Section 6), we present a comprehensive example of taking a sophisticated concurrent data structure, an *elimination stack*, and making it recoverable.

Linked List

Algorithms 1 and 2 show Harris' classic lock-free *linked list* set [12].[1] It supports FIND, INSERT and DELETE the latter two use the SEARCH helper function in order to find the node with the smallest key greater than or equal to the input key (denoted *curr*) and its predecessor in the list (denoted *pred*). DELETE first logically deletes a node by marking it as deleted and then physically removes it from the list. [add here description how this is done, or reference to details in section?!][add a bit more detail]

To find a key k, a process simply looks for an unmarked node with key k (lines 16-19). To insert a key k, process p calls SEARCH in order to find the position in the list where k should be added (line 23) If k is already in the list, INSERT returns **false** (lines 24-25), otherwise it tries to set *pred.next* to point to a new node containing k using *CAS* (line 29); this fails if *pred* has been logically deleted in the interim. To delete a key k, p calls SEARCH returning **false** if k not in the set (lines 32-33). Otherwise, p repeatedly tries to logically delete it by marking the *next* field of its node using *CAS*, until the node is marked (lines 35-37). After the node is marked, p tries to physically remove it (lines 36-39).

The operations are linearized as follows: A *Find* is linearized with the last read of *curr* (line 18), FIND returns **true** if node *curr* is unmarked and **false** otherwise. An *Insert* is linearized either when the SEARCH called by INSERT reads (line 4) an unmarked node with key k, when INSERT returns **false**, or with a successful *CAS* inserting k to the list (line 29), when INSERT returns **true**. A *delete* is linearized either when the SEARCH instance it calls reads in line 4 an unmarked list node with a key

[1] The DELETE pseudo-code is slightly optimized so that if the key is found and is later logically deleted, then DELETE returns **true** if the logical deletion was performed by the current process, and **false** otherwise (lines 75-77, 82).

Algorithm 1: Linked list: SEARCH and FIND

```
Type Node {MarkableNodeRef *next, int key}
Shared variables: Node *head

Procedure ⟨Node *, Node *⟩ SEARCH (T key)
1  Node *pred, *curr, *succ
2  retry: while true do
3      pred := head
4      curr := pred.next
5      while true do
6          succ := curr.next
           // if curr was logically deleted
7          if marked(succ) then
8              if pred.next.CAS
               (< 0, curr >,< 0, succ >) = false then
               // Help physical delete
9                  go to retry            // Help failed
10             curr := succ           // Help succeeded
11         else
12             if curr.key ≥ key then        // First
               unmarked node with key ≥ key
13                 return ⟨pred, curr⟩
14             pred := curr            // Advance pred
15             curr := succ            // Advance curr

Procedure boolean FIND (T key)
16 Node *curr := head
   /* Search for first node with key ≥ key      */
17 while curr.key < key do
18     curr = curr.next
19 return (curr.key = key && ¬marked(curr.next))
```

greater than k (in which case DELETE returns **false**), upon a successful logical deletion by the current operation (in line 37, DELETE returns **true**), or upon a logical deletion of the node by another concurrently executing process (in line 77, DELETE returns **false**).

Thus, successful INSERT and DELETE are linearized when they change the data-structure in a manner visible to other processes, e.g., after the successful *CAS* in line 29 of INSERT, or after a successful logical deletion in line 37 of DELETE. The instructions in lines 29 and 37 are the *realization CAS* for INSERT and DELETE, respectively.

Consider now the situation when some process p recovers from a crash-failure. By the way linearization points are assigned to operations and by the definition of realization *CAS* for INSERT and DELETE, the algorithm satisfies *strict linearizability* [1]: a crashed operation can be either removed from the history or linearized between its invocation and crash. If p failed during an operation *Op*, either the realization *CAS* for *Op* has already been executed in which case the effect of *Op* becomes visible and *Op* is linearized in that *CAS*, or p crashes at some earlier point of *Op*'s exeution in which case

Algorithm 2: Linked list: INSERT and DELETE

Procedure boolean INSERT (T *key*)

```
Node *pred, *curr
Node newnd := new Node (key)
while true do
    // Search the right location to insert
    ⟨pred, curr⟩ := SEARCH(key)
    if curr.key = key then        // key in the list
        return false
    else
        newnd.next := < 0, curr >
        // Try to add newnd
        if pred.next.CAS (< 0, curr >,
        < 0, newnd >) then
            return true
```

Procedure boolean DELETE (T *key*)

```
Node *pred, *curr, *succ     boolean res := false
⟨pred, curr⟩ := SEARCH(key)        // Search for key
if curr.key ≠ key then
    return false              // key not in the list
else
    while ¬marked(curr.next) do    // Repeatedly
    attempt logical delete
        succ := curr.next
        res := curr.next.CAS (< 0, succ >,
        < 1, succ >)
    // Physical deletion attempt
    succ := curr.next
    pred.next.CAS (< 0, curr >, < 0, succ >) return
    res
```

none of the steps *Op* performed before crashing will ever become visible and *Op* is not linearized.

However, the response of the failed operation may be lost. For example, in a scenario in which process p performs DELETE(k) and crashes immediately after applying a *CAS* to mark the node containing key k, p has no way of knowing when it recovers whether its failed operation had any effect on the set. Specifically, p cannot know whether its DELETE has executed its realization *CAS*, and if so what response it should return.

We now explain our approach to make this algorithm recoverable. (See detailed description and code in Section **??**.) Our approach leaves FIND as is, since it is a read-only operation. SEARCH is not read-only, but it can be stopped and restarted at any point, without violating the linearizability of the resulted history, and hence need not be changed.

INSERT is uniquely identify by the node *nd* it tries to add, as different instances creates different nodes. Moreover, once the operation is visible and linearized in a successful realization *CAS*, *nd* can be found in the list, indicating the operation is visible. Finally, *nd* is marked before it is physically removing from the list, indicating that *nd* was in the list. This means there are two indications, one of which holding if a realization *CAS* was successfully performed before the crash, and none of which holds, otherwise.

Our approach makes these operations recoverable, by testing these indications upon recovery. For DELETE, note that *nd* logical deletion (in a successful realization *CAS*) is indicated in *nd* itself, which is marked, and will stay so forever. Therefore, if p crashes during a delete of *nd*, then p can conclude, upon recovery, whether *nd* was deleted by checking if it is marked.

However, p can not tell whether it is the process to delete *nd*. We add a *deleter* field to each node, so that once the node is deleted, the first process to *CAS* its id to *deleter* is the one to perform the DELETE.

Binary Search Tree

In the lock-free *binary search tree* (BSF) of [9], processes help each other to carry their operations. The implementation, described in detail in Section **??**. each operation has an associated info record, initialized in a single CAS by the calling process, and using by helping processes to track their progress.

When a process p needs to modify a node *nd* it marks *nd* with a pointer to its info record; this marking remains until p's operation completes (either by p or by a helping process). This implies that upon recovery after a crash, p can check to see if the node is still marked with its own operation, and if so, try to complete the operation, starting from right after the marking CAS; otherwise, p restarts the operation.

As is often the case when helping is used, several processes may try to perform the same operation (concurrently), leading to redundant efforts that yield only a single completed operation. Even more problematic in the context of the crash-recovery model, and because helping is anonymous, repeated attempts of p to perform an operation (due to crashes) may be indistinguishable from an execution where other processes are helping p complete its operation.

The above recovery scheme may still allow a scenario in which p crashes right after completing its operation and before unmarking the node, causing p to restart the operation upon recovery and applying it twice. To avoid this scenario, a process updates a *done* field in the info record, which signifies the operation is done, before unmarking the node in the cleaning phase. Thus, if the node is not marked with the operation upon recovery, then the new field allows the process to conclude whether the operation has been completed or failed. If a process crashes after updating the done field and before cleaning, then upon recovery, unless another process performed the cleaning, it will observe that the node is marked and complete the cleaning phase.

Elimination Stack

After presenting these specific data structures, linked list and BST, we can explain the two principles underlying our general approach, in the context of a more complex data structure—an *elimination stack* [13].

An elimination stack combines of two components: a central stack and an elimination array. The central stack is implemented in a way that resembles Harris' linked list, described above: to push or pop the process tries to atomically swing the head pointer using CAS. Therefore, the same approach taken in the Linked-List can be used for this case also. The Pop operation will use a new *popby* field, similar to the deleter field, in order to determine which process is the one to pop the node.

However, unlike the linked list, where a node is first marked before removing it from the list, in the stack a Pop operation is done using a single CAS without marking. Hence, if a process p crashes right after successfully pushing a new node and the node is later removed from the [[stack??]], p can not tell between the resulting situation and one in which the Push has failed.

This problem is resolved by having a process mark a node [[as part of the list??]] before trying to pop it; marking does not mean that the node has been removed, as the Pop operation may fail. For efficiency, the *popby* field is used also for this marking. A process first writes NULL to it, and in case of a successful Pop try to write its id. This is all done using CAS, so to avoid overwriting.

In the elimination array each entry holds an *exchanger* object that matches two processes—one doing a push and another doing a pop—to exchange their values. We change the implementation such that a process first creates an info record containing its values, and then processes exchange info records instead of values: The process doing a push writes its info record by a CAS on an exchanger, while the process doing a pop takes this it by replacing it, with a CAS, with its own info record.

To allow recovery, the info record includes s a state, identifying whether p is the first or second process to write to an exchanger, i.e., whether it was performing a push or a pop; in the latter case, the record also contains a reference to the info record p is trying to collide with. If p crashes and the info record indicates it is part of an ongoing exchange, then p tries to complete the exchange; otherwise, either its exchange attempt failed, or that it succeeded and completed by some other process. The latter case is indicated in the info record, which also contains the response value.

The info records also support a simple and efficient helping mechanism. After a process writes its info record second to the exchanger, any other process complete the exchange by reading this record and updating both records with the right response values. That is, other processes do not wait for the two colliding processes and complete the exchange on their behalf. Then, they can reset the exchanger, making it ready for reuse.

3 Model and Definitions

We consider a system where N asynchronous *processes* $p_1, \ldots, p_N$ communicate by applying operations to *concurrent objects*. The system provides *base objects* that support atomic read, write, and read-modify-write primitive operations. Base objects can be used for *implementing* more complex concurrent objects by defining algorithms, for every process, that implement the operations of the implemented object using primitive operations. These more complex objects (together with base objects) may be used in turn, similarly, for implementing even more complex objects, and so on.

The state of each process consists of *non-volatile* shared-memory variables, as well as *local* variables stored in volatile processor registers. At any point during the execution of an operation, a process can incur a *crash-failure* (or simply a *crash*) that resets all its local variables to arbitrary values, but preserves the values of all its non-volatile variables. A process p *invokes an operation* Op on an object by performing an *invocation step*. Upon Op's completion, a *response step* is executed, in which Op's response is stored to a local variable of p. The response value is lost if p incurs a crash before *persisting* it (i.e. before writing it to a non-volatile variable).

Operation Op is *pending* if it was invoked but was not yet completed. For simplicity, we assume that, at any point in time, each process has at most a single pending operation on any single object[2]. An operation Op is called *recoverable* if there is a *recovery function*, denoted Op.`Recover`, associated with it which is responsible for completing Op upon recovery from a crash. If all operations of an implementation of an object O are recoverable, then the implementation is called *recoverable*. The execution of operations (and recoverable operations in particular) may be nested, that is, an operation Op can invoke another operation Op_1. For example, during the execution of a recoverable operation Op on a simulated object O by process p, p may invoke a recoverable operation Op_1 on another object O_1. Consequently, multiple nested invocations of recoverable operations on different objects by process p (in our example, Op and Op_1)

[2]This assumption can be removed, but this would require substantial changes to the notions of sequential executions and linearizability which we chose to avoid in this work.

can be pending at any point in time. Following a crash of process p, the system may eventually resurrect process p by invoking the recovery function of the (inner-most) operation that p was executing at the time of the failure. The invocation of this recovery function comprises a *recovery step for p*.

More formally, a *history* H is a sequence of *steps*. There are four types of steps in a history:

1. an *invocation step*, denoted (INV, p, O, Op), represents the invocation of operation Op on object O by process p;
2. an operation Op can be completed either *directly* or when, following one or more crashes, the execution of the last instance of Op.`Recover` invoked by the system for p is completed. In either case, a *response step s*, denoted (RES, p, O, Op, ret), represents the completion of operation Op invoked on object O by process p. When s takes effect, the response ret is written to a local variable of p. If s' is the invocation step of Op by p, we say that *s matches s'*;
3. a *crash step s for process p*, denoted $(CRASH, p)$, represents the crash of process p. We call the inner-most recoverable operation Op invoked by p that was pending when the crash occurred, the *crashed operation* of s; p may crash also when executing Op.`Recover`, in which case another $(CRASH, p)$ step is appended to H and Op is the crashed operation of s also in this case.
4. a *recovery step s for process p*, denoted (REC, p), is the only step of p that is allowed to follow a $(CRASH, p)$ step s'. It represents the resurrection of p by the system, in which it invokes Op.`Recover`,[3] where Op is the crashed operation of s'. We say that *s is the recovery step that matches s'*.

When a recovery function Op.`Recover` is invoked by the system to recover from a crash, we assume it receives the same arguments as those with which Op was invoked when the crash occurred. We also assume the existence of a per-process non-volatile variable CP_p that may be used by recoverable operations and recovery code for managing check-points in their execution flow. The system stores to CP_p the address of the first instruction of a recoverable operation Op when Op is invoked, and stores to it the address of the next instruction of Op to be executed when a function call invoked by Op returns. CP_p can be read and written by recoverable operations (and their recovery functions)[4]. In what follows, we consider only histories that arise from recoverable implementations.

For a history H, we let $H|p$ denote the subhistory of H consisting of all the steps by process p. We let $H|O$ denote the subhistory of H consisting of all the invocation and response steps on object O in H, as well as any crash step in H, by any process p, whose crashed operation is an operation on O and the corresponding recover step by p (if it appears in H) H is *crash-free* if it contains no crash steps (hence also no recovery steps). Let $H|{<}p, O{>} = (H|O)|p$. A crash-free subhistory $H|O$ is well-formed, if for every process p, $H|{<}p, O{>}$ is a sequence of alternating matching invocation and response steps, starting with an invocation step.

Given two operations op_1 and op_2 in a history H, we say that op_1 *happens before* op_2, denoted by $op_1 <_H op_2$, if op_1 has a response step in H that precedes the invocation step of op_2 in H. If neither $op_1 <_H op_2$ nor $op_2 <_H op_1$ holds, then we say that op_1 and op_2 are *concurrent* in H. $H|O$ is a *sequential object history*, if it is an alternating sequence of invocations and their matching responses starting with an invocation (the sequence may end with a pending invocation). The *sequential specification* of an object O is the set of all possible (legal) sequential histories over O.

A crash-free history H is *well-formed* if: 1) for *every* object O, $H|O$ is well-formed, and 2) for *every* process p, and for every two pairs $\langle i_1, r_1\rangle$ and $\langle i_2, r_2\rangle$ of matching invocation/response steps in $H|p$ such that i_1 precedes i_2 and i_2 precedes r_1, then it holds that r_2 precedes r_1. The second requirement guarantees that if process p, while executing an operation Op_1, invokes an operation Op_2, Op_2's response must precede Op_1's response. In what follows, we consider only well-formed histories.

Two histories H and H' are *equivalent*, if $H|p = H'|p$ for all processes p. A history H is *object-sequential*, if $H|O$ is sequential for all objects O that appear in H. Given a *finite* history H, a *completion* of H is a history H' constructed from H by selecting separately, for each object O that appears in H, a subset of the operations pending on O in H and appending matching responses to all these operations, and then removing all remaining pending operations on O (if any).

Definition 1 (Linearizability [15], rephrased)**.** *A finite crash-free history H is* linearizable *if there exists a completion H' of H and an object-sequential history S such that the following requirements hold: (i) H' is equivalent to S, (ii) $S|O$ is legal for all objects O, and (iii) $<_{H'} \subseteq <_S$ (i.e., if $op_1 <_{H'} op_2$, then $op_1 <_S op_2$).*

Thus, a finite history is linearizable, if we can linearize the subhistory of each object that appears in it. Next, we define a more general notion of well-formedness

[3] A history does not contain invocation/response steps for recovery functions.

[4] This is a relaxation of the model of [2], which assumed that recovery code has access to the address of Op's instruction that p was about to execute when it crashed.

that applies also to histories that contain crash/recovery steps. For a history H, we let $N(H)$ denote the history obtained from H by removing all crash and recovery steps.

Definition 2. *A history H is recoverable well-formed if (i) $N(H)$ is well-formed, and (ii) every crash step in $H|p$ is either p's last step in H or is followed in $H|p$ by a matching recovery step of p.*

Definition 3. *A finite history H satisfies* nesting-safe recoverable linearizability *(NRL) if it is recoverable well-formed and $N(H)$ is a linearizable history. An object implementation satisfies NRL if every history it produces satisfies NRL.*

4 Linked-List Based Set

In this section, we present a recoverable version of the linked-list set algorithm of Harris [12].[5]

The linked-list set supports the Find, Insert and Delete operations. The algorithm maintains a linked list of nodes sorted in increasing order of keys. The list always contains a *head* and *tail* sentinel nodes, containing keys $-\infty, \infty$, respectively. The pseudo-code for the algorithm is presented in Algorithms 3-4. Pseudo-code in blue font was added for recoverability. the *next* field of each node consists of a reference to the next node and a *marked* bit that is set when the node is logically deleted. Both components can be manipulated atomically, either together or individually, using a singe-word *CAS* operation[6]. The *marked* predicate can be applied to the next field of a node to determine whether or not the node is marked.

We now describe an extension of the Harris algorithm that allows a process recovering from a crash-failure to determine whether its failed operation completed, and return the correct response. The recovery function of the Find operation simply re-invokes Find (hence its pseudo-code is not shown). To support the recovery of Insert and Delete operations, a (persistent) shared-memory array *RD* was added, where *RD*[p] stores recovery data for process p. More specifically, variable *RD*[p] contains a pointer to an Info structure storing recovery data for the process' current recoverable operation. Each Info structure stores two fields - a reference *nd* to a node and a *result* field used for persisting the response value for the operation before returing.

We now describe the additions to the Insert operation that were introduced for supporting recoverability. First, Insert installs a fresh Info structure into *RD* in line 43, whose *nd* field points to a newly allocated node structure (the *deleter* field is only used by Delete operations and is described later). It then updates p's check-point variable (in line 44) by invoking the curPC macro, returning the current value of the program counter (for simplicity, we assume that it is 44, in this case). By executing this line, p *persistently reports* that the info structure for its current operation has been installed. Finally, once Insert determines its response, it persists it just before returning (in lines 48 and 53).

We now describe the Insert.Recover function. Let W denote the instance of Insert from whose failure Insert.Recover attempts to recover. Insert.Recover starts by reading p's check-point variable in line 56 in order to check whether p's current info structure was installed by W. If the failure occurred before W persistently reported that it installed its info structure (in line 44), then W is re-executed from scratch. Otherwise, the following actions take place. If a response was already written to W's Info structure, then Insert.Recover simply returns this response (lines 58-59). We are left with the case that a response was not yet written by W to its Info structure, so either W did not execute a successful realization *CAS* (in line 52) or its realization *CAS* succeeded but W failed before writing the response in line 53. In order to determine which of these two scenarios occurred, Insert.Recover searches the list for *key* (in line 60). If either the key is found in the list inside the node allocated by W or if that node was marked for deletion (line 61), then W had executed its realization *CAS* before the failure occured (i.e. W succeeded in inserting its node in the linked list), and so the recovery function persists the response and returns **true**(lines 62-63). Notice that indeed if the node has been successfully linked in the list, the only way for the Search not to find it is if it has, in the meantime, been deleted. However, in that case, the node will have, first, been marked for deletion, and therefore the condition of the if statement of line 61 will be evaluated to true. Otherwise, recovery proceeds from line 40 in order to re-attempt insertion.[7]

Next, we describe how recoverability is ensured for Delete operations. Similarly to the recoverable Insert operation, a recoverable Delete first installs a fresh Info

[5]Some implementation details follow the algorithm's presentation in [14].

[6]In the Java implementation of the algorithm presented in [14], *next* fields are represented by AtomicMarkableReference objects. In the implementation of an AtomicMarkableReference object the marked bit occupies the least significant bit of the reference.

[7]BY "recovery proceeds from line X" we do not mean to say that there is a jump to line X, since in order to do so execution context before line X must be saved and then restored. Instead, we mean that the pseudo-code performed by the recovery function from this point on is the same as that of the recoverable operation starting from line X. One efficient way of implementing this is to have both the recoverable operation and the recovery function invoke a parameterless macro call that embeds this pseudo-code during compilation pre-processing.

structure into *RD* in line 66. It then updates p's check-point variable (in line 68) to *persistently report* that its info structure has been installed. If *key* is not found (line 69), the response (which is in this case **false**) is persisted and then it is returned (lines 71-72). If *key is* found (in node *curr*), a reference to *curr* is persisted to the *nd* field of p's Info structure. Following this, p proceeds as in the original algorithm by repeatedly trying to mark *curr* using *CAS* in lines 75-77 (i.e. it repeatedly executes *CAS* until it logically deletes the node). Once it is marked, p tries to physically remove *curr* in lines 78-79.

The key technical difficulty for supporting recoverability of DELETE operations is the following. If p fails immediately before or after the *CAS* of line 77, recovers and then finds that *curr* was logically deleted, how can it know whether it was the (single) process that succeeded in deleting *curr*? Our solution to this problem is to "attribute" a node's deletion to a single process using a new node-field called *deleter*, initialized to ⊥. After p finds that *curr* is logically deleted (marked) in line 75, *regardless of whether it was marked by p or by another process*, p tries to establish itself as the deleter of *curr* by atomically changing *curr.deleter* from ⊥ to its ID using *CAS* (line 81). Finally, p persists and returns the result of this *CAS* as the response of the DELETE operation in lines 81-82.

We now describe the DELETE.RECOVER function. Let D denote the instance of DELETE from whose failure DELETE.RECOVER attempts to recover. DELETE.RECOVER starts by reading p's check-point variable in line 84. If the failure occurred before D persistently reported that it installed its info structure (in line 68), then D is re-executed from scratch. Otherwise, the following actions take place. If a response was already written to D's Info structure, then DELETE simply returns this response (lines 86-87). Otherwise, if the *nd* field of p's Info structure was previously set and node *nd* is logically deleted (line 88), p attempts to establish itself as the deleter of *nd* using *CAS*, and then persists and returns the result according to the id written in *curr.deleter* (lines 89-92) as the response of its DELETE operation. Finally, if the condition of line 88 does not hold, p re-attempts the deletion (line 94).

4.1 Correctness Argument

In the following, we give a high-level argument for the correctness of the algorithm. We say that a node is in the linked-list if it is reachable by following *next* pointers starting from *head*. We say that a node is in the implemented set if it is in the linked-list and is not marked.

The proof relies on the following observations. As long as a node *nd* is in the linked-list there is exactly one node in the list pointing to it. In addition, *nd* can be marked exactly once, and it stays so forever, as any *CAS* is executed with an unmarked node as its first argument. For the same reason, *nd* can be physically deleted exactly once, since no node in the list points to it once it is deleted, and we never add a marked node back to the list.

FIND operation is implemented in a read-only manner, and thus it can be re-executed without effecting any other concurrent operation. In addition, SEARCH routine, even though not read-only, simply traverses the list while trying to physically delete any marked node it encounters. As any marked node can be physically deleted exactly once, re-executing SEARCH can not cause a deletion of an unmarked node, or a deletion of the same node more then once. This implies re-executing SEARCH can only help physical deletion of more nodes, and does not effect the list in any other way.

INSERT and DELETE first install info structure and set a check-point. Clearly, a crash before the check-point implies the operation did not effect the list or any other operation, and in such case the RECOVER function simply re-execute the operation. This argument holds for any number of crashes, as long as the check-point is yet to be set.

Assume process p performs an INSERT(key) operation. If p does not crash, then it repeatedly search for the right location for the new node *newnd*, and tries to insert it. In case p updates $RD[p].result$ to **false** in line 48, then the preceding SEARCH in line 46 finds a node *curr* in the set with data *key* (when SEARCH read *curr* it is not marked). Therefore, there is a point along the INSERT operation where the set contains *key*, and the operation is linearized at this point. If p crash after updating *result* then eventually, in order to complete INSERT.RECOVER p must read $RD[p].result$ and return **false**.

In case p performs its realization *CAS* in line 52, then *newnd* is in the set. The only way to physically remove it from the list is by first marking it. Therefore, after the realization *CAS*, either *newnd* is in the list, or it is marked, and one of these conditions must hold. As a result, in any crash after the realization *CAS* the INSERT.RECOVER function returns **true**, either in line 59 (because *result* has already been updated), or in line 61 which evaluates to **true**. In particular, each INSERT operation can perform at most a single realization *CAS*, as any crash after it results a response of **true**, without performing any more *CAS* instances.

We are left with the case where p crash before performing its realization *CAS* or updating *result*. In such case, *newnd* was not added to the list yet, and in particular no process can mark it. As a result, INSERT did not effect any other operation, and indeed INSERT.RECOVER re-executes it.

Assume process p performs DELETE(key) operation. If p updates $result$ to **false** in line 71, then the preceding SEARCH found two adjustments nodes $pred$ and $curr$ in the list, where $pred.key < key < curr.key$. Since we keep the list sorter in an increasing manner, it follows the list does not contains key at the point when $pred$ points to $curr$. In particular, there is point along the interval of DELETE where key is not in the set, and the operation is linearized at this point. Any crash after line 71 results the DELETE.RECOVER function must read $result$ and return **false**.

Assume now p does not write in line 71. A node $curr$ is written to $RD[p].nd$ in line 74 only if SEARCH observes $curr$ is in the list and not marked. Clearly, a crash before updating $RD[p].nd$ implies the operation did not mark any node, nor effected any other operation, and DELETE.RECOVER simply re-execute DELETE. On the other hand, once p updates $RD[p].nd$, it keeps trying to mark nd. If p crash and recovers, and finds nd is not marked (line 88), then in particular p's operation did not mark nd, nor effected the list or any other operation. In such case, DELETE.RECOVER re-executes DELETE. This argument holds for any number of crash and recover, as long as p observe nd is not marked.

If p observes nd is marked, either in the DELETE or in DELETE.RECOVER, we conclude the marking was done after the read of nd in SEARCH. Moreover, once nd is marked, in order to complete the DELETE operation, p must performs CAS, trying to write its name to $nd.deleter$, either in DELETE, or in DELETE.RECOVER. Since $deleter$ is initialised to $\perp$, only the first such CAS succeeds. Let q be the first process to perform such CAS, if exists. Then, it set $nd.deleter$ to q, and this does not change. Once $deleter$ is set, if $p = q$, then it can only return **true**, even in case it crash, as line 90 always evaluates to **true**. Otherwise, $p \neq q$, and by similar argument it can only return **false**. As a result, any process trying to delete nd and observe it is marked (at any point), must have the marking step in its DELETE operation interval. In addition, exactly one such process, denoted q, returns **true**, while any other returns **false**. We linearize the DELETE operation of q at the time of the marking, and any other DELETE returning **false** is linearized right after it (in an arbitrary order).

Algorithm 3: Recoverable linked list: INSERT

```
Type Node {MarkableNodeRef *next, int key, int deleter}
Type Info {Node *nd, boolean result}
Shared variables: Node *head, Info* RD[N]

Procedure boolean INSERT (T key)
40 Node *pred, *curr
41 Node newnd := new Node (key)
42 newnd.deleter := ⊥
43 RD[pid] := new Info (newnd, ⊥)      // Install Info structure for this operation
44 CP_p := curPC()      // Set a check-point indicating Info structure was installed
45 while true do
       // Search for right location to insert
46     ⟨pred, curr⟩ := SEARCH(key)
47     if curr.key = key then      // key in the list
48         RD[pid].result := false      // Persist response
49         return false
50     else
51         newnd.next := < 0, curr >
           // Try to add newnd
52         if pred.next.CAS (< 0, curr >, < 0, newnd >) then
53             RD[pid].result := true      // Persist response
54             return true

Procedure boolean INSERT.RECOVER (T key)
55 Node *nd := RD[pid].nd
   // Failed before installing info structure
56 if CP_p < 44 then
57     Proceed from line 40      // re-execute
   // If operation response was persisted
58 else if RD[pid].result ≠ ⊥ then
59     return RD[pid].result      // Return response
60 ⟨pred, curr⟩ := SEARCH(key)      // Search for nd in the list
   // If nd in list or is marked (hence was)
61 if curr = nd || marked(nd.next) then
62     RD[pid].result := true      // Persist response
63     return true
64 else
65     Proceed from line 40      // Re-attempt insertion
```

5 Robust BST

The original BST algorithm does not support the crash-recovery model. It is clear from the code a process does not persist the operation's response in the non-volatile memory, and thus, once a process crash the response is lost. For example, assume a process q apply INSERT(k), performs a successful CAS in line **??** and fails after completing the HELPINSERT routine. In this case, the INSERT operation took effect, that is, the new key appears as a leaf in the tree, and any FIND(k) operation will return it. However, even though the operation must be linearized before the crash, upon recovery process q is unaware of it. Moreover, looking for the new leaf in the tree may be futile, as it might be k has been removed from the tree after the crash.

Furthermore, if no recover routine is supplied, it may result an execution which is not well-formed. Consider for example the following scenario. A process q invoke

Algorithm 4: Recoverable linked list: DELETE

```
Procedure boolean DELETE (T key)
66 Node *pred, *curr, *succ     boolean res := false
67 RD[pid] := new Info (⊥, ⊥)          // Install info
   structure for this operation
68 CP_p := curPC()      // Set a check-point indicating
   Info structure was installed
   // Search for key in the list
69 ⟨pred, curr⟩ := SEARCH(key)
70 if curr.key ≠ key then      // key not in the list
71 |   RD[pid].result := false     // Persist response
72 |   return false
73 else
74 |   RD[pid].nd := curr     // Persist reference to
   |   node containing key
75 |   while ¬marked(curr.next) do     // Repeatedly
   |   attempt logical delete
76 |   |   succ := curr.next
77 |   |   res := curr.next.CAS (< 0, succ >,
   |   └   < 1, succ >)
   |   // Physical deletion attempt
78 |   succ := curr.next
79 |   pred.next.CAS (< 0, curr >, < 0, succ >)
80 |   // Try establishing yourself as deleter
81 |   res := curr.deleter.CAS(⊥, pid)
   |   RD[pid].result := res       // Persist response
82 |   return res

Procedure boolean DELETE.RECOVER (T key)
83 Node *nd := RD[pid].nd ,     boolean res:=false
   // Failed before installing info structure
84 if CP_p < 68 then
85 |   Proceed from line 66                 // re-execute
   // If operation response was persisted
86 else if RD[pid].result ≠ ⊥ then
87 |   return RD[pid].result      // Return response
   // If nd was logically deleted
88 if nd ≠ ⊥ && marked(nd.next) then
   |   // Try establishing yourself as deleter
89 |   nd.deleter.CAS (⊥, pid)
90 |   res := (nd.deleter == pid)
91 |   RD[pid].result := res       // Persist response
92 |   return res
93 else
94 |   Proceed from line 66       // Re-attempt deletion
```

an Op_1 = INSERT(k_1) operation. q performs a successful CAS in line ?? followed by a crush. After recovering, q invoke an Op_2 = INSERT(k_2) operation. Assume k_1 and k_2 belongs to a different parts of the tree (do not share parent or grandparent). Then, q can complete the insertion of k_2 without having any affect on k_1. Now, a process q' performs FIND(k_1) which returns NULL, as the insertion of k_1 is not completed, followed by FIND(k_2), which returns the leaf of k_2. The INSERT(k_1) operation will be completed later by any INSERT or DELETE operation which needs to make changes to the flagged node. We get that Op_2 must be before Op_1 in the linearization, although Op_1 invoked first.

The kind of anomaly described above can be addressed by having the first CAS of a successful attempt for INSERT or DELETE as the linearization point, as in the Linked-List. For that, the FIND routine should take into consideration future unavoidable changes, for example, a node flagged with IFlag ensures an insertion of some key. A simple solution is to change the FIND routine such that it also helps other operations, as described in figure ??. The FIND routine will search for key k in the tree. If the SEARCH routine returns a grandparent or a parent that is flagged, then it might be that an insert or delete of k is currently in progress, thus we first help the operation to complete, and then search for k again. Otherwise, if *gpupdate* or *pupdate* has been changed since the last read, it means some change already took affect, and there is a need to search for k again. If none of the above holds, there is a point in time where gp points to p which points to l, and there is no attempt to change this part of the tree. As a result, if k is in the tree at this point, it must be in l, and the find can return safely.

The approach described above is not efficient in terms of time. We would like a solution which maintain the desirable behaviour of the original FIND routine, where a single SEARCH is needed. A more refined solution is given in figure ??. The intuition for it is drawn from the Linked-List algorithm. In the Linked-List algorithm it was enough to consider a marked node as if it has been deleted, without the need to complete the deletion. Nonetheless, the complex BST implementation is more challenging, as the DELETE routine needs to successfully capture two nodes using CAS in order to complete the deletion. Therefore, if a process p executes FIND(k) procedure, and observes a node flagged with DFlag attempting to delete the key k, it can not know whether in the future this delete attempt will succeed or fail, and thus does not know whether to consider the key k as part of the tree or not. To overcome this problem, in such case the process will first try and validate the delete operation by marking the relevant node. According to whether the marking attempt was successful, the process can conclude if the delete operation is successful or not. In order to easily implement the modified FIND routine there is a need to conclude from IInfo what is the new leaf (leaf *new* in the INSERT routine). For simplicity of presentation, we do not add this field, and abstractly refer to it in the code.

The correctness of the two suggested solutions relies on the following argument. Once a process flags a node during operation *Op* with input key *k* (either INSERT or DELETE), then if this attempt to complete the operation eventually succeed (i.e., the marking is also successful in the case of DELETE), then any FIND(k) operation invoked from this point consider *Op* as if it is completed.

The suggested modification, although being simple and local, only guarantee the implementation satisfy R-linearzability. However, the problem of response being lost in case of a crash is not addressed. Roughly speaking, the critical points in the code for recovery are the CAS primitives, as a crash right after applying CAS operation results the lost of the response, and in order to complete the operation the process needs to know the result of the CAS. In addition, because of the helping mechanism, a suspended DELETE operation which flagged a node and yet to mark one, may be completed by other process in the future, and may not. Upon recovery, the process needs to distinguish between the two cases, in order to obtain the right response.

To address this issue, we expend the helping mechanism so that it also update the info structure in case of a success. This is done by adding a boolean field, done, to the Flag structure. This way, if a process crash along an operation *Op*, upon recovery it can check to see if the operation was already completed. A crucial point is to update the *done* bit before performing the unflagging. Therefore, if a node is no longer flagged we can be sure done was already updated. If we switch the order, then it might be an operation and unflagging were completed, but the done bit is yet to be updated. Therefore, other processes can change the BST structure. However, if the process crash and recover at this point, the done bit is off, and the BST structure has been changed, so it will be harder for the process to conclude whether the operation took affect.

Before a process *q* attempt to perform an operation, as it creates the Flag structure *op* describing the operation and its affect on the data structure, the process stores *op* in a designated location in the shared memory (for simplicity, we use an array). As a result, upon recovery *q* has an access to this information. Now, *q* can check to see if the operation is still in progress, i.e., if the relevant node (parent or grandparent) is still flagged. If so, it first tries to complete the operation. Otherwise, it implies either the operation was completed, and therefore done bit is updated, or that the attempt was unsuccessful and there wa no write to the done bit. Hence, the done bit can distinguish between the two scenarios. Notice that there is a scenario in which process *q* recovers and observes an operation *Op* as it in a progress, but just before it retries it, some other process complete the operation. We need to prove that even in such case, the operation will affect the data structure exactly once, and the right response is returned.

The given implementation does not recover the FIND routine, since this routine does not make any changes to the BST, hence it is always safe to consider it as having no linearization point and reissue it. Also, for ease of presentation, we only write to $Announce[id]$ once we are about to capture a node using a CAS. However, writing to $Announce[id]$ at the beginning of the routine may be helpful in case of a crash early in the routine, so that the process will be able to use the data stored in $Announce[id]$ in such case also. The same is true with response value, $Announce[id] \rightarrow done$ is updated only if the routine made changes to the BST.

Correctness Arguments

In the following section we give a proof sketch for the algorithm correctness. We assume for simplicity nodes and Flag records are always allocated new memory locations, although it is enough to require no location is reallocated as long as there is a chain of pointers leading to it. The proof relies on the correctness of the original algorithm, which can be found on [....].

The proof relies on several key arguments given below.

[**Arg1**] The original algorithm is anonymous and uniform, i.e., any number of processes can use the BST, and there is no need to know the number of processes in the system in order to use the BST. Notice that all helping routines in the given implementation are completely anonymous, and an execution of such a routine by either the process which invoked *op* or any other helping process executes the exact same code. This observation allows the use of the following argument. If a process crash while executing some helping routine, we can consider it as an helping process which stop taking steps (more formally, there is an equivalent execution in which there is such a process, and it is indistinguishable to all process in the system). Since such process can not cause a wrong behaviour of the algorithm, so does the crash. A corollary of this argument is that repeating an helping routine multiple times by the same process can not violate the BST specification, as there is an equivalent executions in which multiple processes executes the different helping routines.

[**Arg2**] It is easy to verify the post-conditions of the SEARCH routine still holds, as they follow directly from the routine's code, and does not rely on the structure or correctness of the BST. Also, the SEARCH routine does not make any changes to the BST, but rather simply traverse it. Therefore FIND routine, which only uses

Search, does not affect any process, and in case of a failure along Find execution, reissuing it satisfies NRL.

[**Arg3**] If an internal node nd_1 stops pointing to a node nd_2 at some point of the execution, it can not point to nd_2 again. This attributes to the fact an Insert presents a node with two new children. Therefore, if nd_2 is a leaf, it can either be delete, or replaced by a new copy of an Insert operation. Otherwise, nd_2 is an internal node, and as such, the pointer to it by nd_1 can not be replaced by an Insert operation (which only allows to replacement a leaf), and therefore it can only be removed from the tree.

[**Arg4**] The field update of a node *nd* can have any value only once along an execution. Any attempt to perform an operation creates a new record in the memory. If $nd \to update$ is marked, it can not be unmarked or changed. Otherwise, any attempt to flag it uses a new created record *op*. If the attempt succeed, then eventually it will be unflagged while still referring to *op*. In order to replace the value again, there must be an operation reading $nd \to update$ after it was unflagged (as any operation first help a flagged node). This operation must create a new record, and thus we can use the same argument again. As a corollary, if a process successfully flag or mark a node, there was no change to the node since the last time it read the update field of the node.

Proof Sketch Assume a process q performs an operation Op (either Insert or Delete). If q does not crash, the algorithm is identical to the original algorithm, except for the additional write to $Announce[q]$ and $op \to done$, and thus the correctness of the original algorithm can be applied. Otherwise, q crash at some point, and upon recovery it reads *op* from $Announce[q]$. This record represent the last attempt of q to complete Op. We split the proof based on the type of operation.

Op = Insert. Consider the read of $op \to p \to update$ upon recovery, and denote this value by *pupdate*. If $pupdate = \langle \text{IFlag}, op \rangle$, this implies the iflag CAS in line **??** was successful and the operation is yet to complete. It might be that Insert already took affect, that is, the new key is part of the tree, but the unflagging is yet to happen. In such case, q calls HelpInsert(op) in order to try and complete the operation. Considering arg1, this call can not violate the BST correctness, even if it not the first time q executes it. Moreover, during HelpInsert there is a write to $op \to done$, and thus after completing the routine q returns True, as required.

Else $pupdate \neq \langle \text{IFlag}, op \rangle$. There are two scenarios to consider. Either the iflag CAS of q in line **??** was successful or not. If it was successful, then $p \to update = \langle \text{IFlag}, op \rangle$ at this point. The only way to change it is to first unflag p. To do so, a process needs to complete an HelpInsert(op) routine, and in particular must write to $op \to done$. In such case, the Insert operation was completed, and q returns True. Otherwise, the CAS was not successful, either because it failed, or the crash was before the CAS. In both cases, the Insert operation will not be completed, as *op* is not stored in $p \to update$, and thus no process has an access to it. Consequently, no process can update $op \to done$, and q returns Fail.

Op = Delete. Consider the read of $op \to gp \to update$ upon recovery, and denote this value by *gpupdate*. If $gpupdate = \langle \text{DFlag}, op \rangle$, this implies the dflag CAS of q in line **??** was successful, and the operation is yet to complete. As in the Insert, it might be the operation already changed the tree. After reading *gpupdate* q invokes HelpDelete(op) routine. Again, following arg1, executing this multiple times by q can not violate the BST correctness. The first process to try and mark $op \to p \to update$ during an HelpDelete(op) routine is the one to determine the outcome of it. If it is successful, then p is marked, and the *update* field can not be changed. That is, any HelpDelete(op) execution will obtain true in line **??**, and will call HelpMarked(op) routine. Otherwise, the CAS fails, and so $p \to update$ is no longer equal to $op \to pupdate$. By arg4 it will never get this value again, and thus any marking CAS during a HelpDelete(op) execution will fail, and there is no call to HelpMarked(op). In the first case, any HelpDelete(op) routine must first complete a HelpMarked(op), and thus must write to $op \to done$, while in the later case, there is no write to $op \to done$, as no HelpMarked(op) is ever invoked. Therefore, in both cases, when q completes HelpMarked(op) it reads $op \to done$ and returns the right response.

Otherwise $gpupdate \neq \langle \text{DFlag}, op \rangle$, and there are two scenarios to consider. If the dflag CAS of q in line **??** never took affect, because it either failed, or the crash preceded it, then *op* is never written to $gp \to update$, or to any update field. Thus, no process is aware of it, and $op \to done$ remains False, resulting q returning Fail as required. Else, the CAS was successful, and $gp \to update$ was flagged. The only way to change it is to first unflag it, and this in turn can be done only during an HelpDelete(op) routine. In this case, it can be unflagged in either the HelpMarked routine in line **??**, or in line **??** of the HelpDelete routine. As mention before, the first CAS in line **??** of an HelpDelete(op) execution determines the outcome for all HelpDelete(op). If it is successful, $p \to update$ is forever marked, and all HelpDelete(op) must invoke HelpMarked(op). Therefore, the only option to unflag $gp \to update$ is at the end of HelpMarked(op) routine, and this done only after setting $op \to done$. In such case, the Delete operation took affect, and q will return True. On the other hand, if the CAS was not successful, then any HelpDelete(op)

will fail to mark $p \rightarrow update$, and hence no HelpMarked(op) is ever invoked. As a result, there is no write to $op \rightarrow done$. In such case, the Delete operation did not took affect, nor will be, and indeed q will return Fail.

6 Elimination Stack

For simplicity, we assume a value $\perp$, which is different from Null and any other value the stack can store. Since Null is used as a legit return value, representing the value of Pop operation (when exchanging values using the elimination array), Null can not be used to represent an initialization value, different then any stack value. The same holds for a Node, since a Null node represent an empty stack, the value $\perp$ is used to distinguish between initialization value and empty stack.

For simplicity, we split the Recover routine into subroutines, based on which operation (Push, Pop, Exchange) is pending, or needs to be recover. This can be concluded easily by the type of record stored in *Announce*[*pid*] (ExInfo or OpInfo), thus there is no need to explicitly know where exactly in the code the crash took place. Also, the Recover routine returns Fail in case the last pending operation did not took affect (no linearization point), nor it will take in any future run. In such case, the user has the option to either re-invoke the operation, or to skip it, depends on the needs and circumstances of the specific use of the data structure.

The given implementation ignores the log of failures and successes of the exchange routine when recovering. That is, in case of a crash during an Exchange, a process is able to recover the Exchange routine, however, the log of successes and failures is not update, since it might be the process already updated it. In addition, in case of a Fail response, we do not know whether the time limit (timeout) was reached, or that the process simply crashed earlier in the routine without completing it. The given implementation can be expanded to also consider the log. Nonetheless, for ease of presentation we do not handle the log in case of a crash. Assuming crash events are rare, the log still gives a roughly good approximation to the number of failures and successes, thus our approach might be useful in practice.

6.1 A Lock-Free Exchanger

An exchanger object supports the Exchange procedure, which allows exactly two processes to exchange values. If process A calls the Exchange with argument a, and B calls the Exchange of the same object with argument b, then A's call will return value b and vice versa.

On the original algorithm [cite the book?!], processes race to win the exchanger using a *CAS* primitive. A process accessing the exchanger first reads its content, and act according to the state of it. The first process observe an Empty state, and tries to atomically writes its value and change the state to Waiting. In such case, it spins and wait for the second process to arrive. The second, observing the state is now Waiting, tries to write its value and change the state to Busy. This way, it informs the first one a successful collision took place. Once the first process notice the collision, it reads the other process value and release the exchanger by setting it back to Empty. In order to avoid an unbounded waiting, if a second process does not show up, the call eventually timeout, and the process release the exchanger and return.

Assume a process p successfuly capture the exchanger by setting its status to Waiting, followed by a crash. Now, some other process q complete the exchange by setting the exchanger to Busy. Upon recovery, p can conclude some exchange was completed, but it can not tell whether its value is part of the exchange, and thus it can not complete the operation. Moreover, p and q must agree, otherwise q will return p's value, and thus the operation of p must be linearized together with q operation.

In order to avoid the above problem, we take an approach resembling the BST implementation. Instead of writing a value to the exchanger, processes will use an info record, containing the relevant information for the exchange. This way, processes use the exchanger in order to exchange info records (more precisely, pointers to such records), and not values. To overcome the problematic scenario described earlier, if a process q observe the exchanger state is Waiting with some record *yourop*, it first update its own record *myop* it is about to try and collide with *yourop*, and only then performs the *CAS*. This way, if the collision is successful, the record *myop* which now stored in the exchanger implies which two records collide. Also, the fact that different processes uses different records guarantee that at most one record can collide with *yourop*.

Using records instead of values, when using wisely, allows us to farther improve the algorithm. First, there is no need to store the exchanger's state in it (by using 2 bits of it to mark the state), but we can rather have this info in the record. Second, if there is a Busy record in the exchanger, it contains the info of the two colliding records. Therefore, a third process, trying to also use the exchanger, can help the processes to complete the collision, and then can try and set the exchanger back to Empty, so it can use it again. In the original implementation, a process observaing a Busy exchanger, have to wait for the first process to read the value and release the exchanger. Therefore, if the first process crash after the collision, the exchanger will be hold by it forever.

The helping mechanism avoids this scenario, making the exchange routine non-blocking.

Notice that no exchange record with EMPTY state is ever created, except for the $default$ record. Therefore, reading EMPTY state is equivalent to the exchanger storing a pointer to $default$. A process p creates a new record *myop* when accessing the exchanger, with a unique address. As long as p fails to perform a successful *CAS*, and thus fails to store *myop* in *slot*, it is allowed to try again. However, once a process performs a successful *CAS* and stores *myop* in *slot*, the only other *CAS* it is allowed to do are in order to try and store $defualt$ in $slot$. Thus, *myop* can be written exactly once to $slot$. It follows that a collision can occur between two processes exactly - once a WAITING record stored in *slot*, only a single *CAS* can replace it with a BUSY record. As the two records can not be written again to *slot*, no other process can collide with any of the records.

The EXCHANGE-RECOVER routine relies on the following argument. If a process p successfully wrote op_p to *slot* using the *CAS* in line 106, the only way to overwrite it by a different process q, is by a *CAS* in line 126 with a record op_q such that its state is BUSY, and $op_q.partner = op_p$. In addition, the only way to overwrite op_q is by a *CAS* replacing it with $default$, and this is done only after SWITCHPAIR(op_p, op_q) is completed, and thus both $result$ fields are updated.

The correctness of the EXCHANGE-RECOVER routine is based on the above argument. There are few scenarios to consider. If p crash after a successful *CAS* in line 106, then op_p state is WAITING. Therefore, when reading *slot* in the EXCHANGE-RECOVER one of the following must hold. If *slot* contains op_p, then no process collide with p, and p continue to run as if the time limit has been reached. Otherwise, there was a collision. From the above argument, it must be that either op_q that collide with op_p is stored in *slot*, in this case $op_q.partner = op_p$, and p will try to complete the collision and release *slot*, or that op_q has been overwritten, and in this case the $result$ field of op_p is updated. In both cases, p returns $op_p.result$. If p crash after a successful *CAS* in line 126, then op_p state is BUSY. It follows from the argument that the only way to overwrite op_p is only after completing the collision by SWITCHPAIR. Thus, either upon recovery p reads op_p from *slot*, and in this case it tries to complete the the operation, or that $op_p.result$ was already updated. In both cases, p returns it. If non of the above holds, then op_p was not involved in any collision, because either no successful *CAS* was done by p, or p reached the time limit while no process show up, and was able to set *slot* back to $defualt$. In any case, after the crash of p, op_p will never be written again to *slot*, nor any other op_q such that $op_q.partner = op_p$, as any such op_q tries to perform $CAS(op_p, op_q)$ that will fail. Also, as no process can collide with op_p, no SWITCHPAIR with op_p as parameter is ever invoked, and in particular $op_p.result = \perp$ for the rest of the execution. This in turn implies that upon recovery p will return FAIL, as required.

6.2 Lock-Free Stack

The stack implementation is due to [....]. The TRYPUSH routine tries to atomically have a new node pointing to the old top, and then updating the top to be the new node. The TRYPOP routine tries to atomically read the top of the stack, and change the top to the next node of it. The two routines uses *CAS* in order to gurantee no change for the top was made between the read and write. PUSH (resp. POP) routine is alternating between a TRYPUSH (TRYPOP) routine, which access the central stack, and the EXCHANGE, trying to collide with an opposite operation.

In order to make the implementation recoverable, we need a way to infer whether a POP or PUSH already took affect, in case of a crash. Moreover, in case of a POP, we also need to infer which process is the one to pop the node. For that, we use an approach similar to the Linked-List implementation. Each node contains a new field *popby* which is used to identify a PUSH of the node completed, as well as a POP of the node was completed, and who is the process to pop it. Consider the following scenario. Assume a process p performs a PUSH operation with node *nd*, and using a *CAS* succeed to update the stack top to point to *nd*, followed by a crash. Now, process q performing a POP operation performs a *CAS* causing the removal of *nd* from the stack (by changing top to the next node). In this case, once p recovers, *nd* is no longer part of the stack, and it is also not marked as deleted. This is indistinguishable from a configuration in which the PUSH of *nd* was yet to take affect (a crash before *CAS*), and thus p can not know what the right response is.

One way to solve this issue is by first marking a node for removal, and only then remove it. This way, if a node is no longer part of the stack it must be marked, and thus we can conclude it was in the stack, and the PUSH routine was successful. However, such an implementation, in addition for the need of to system to support a markable reference, also requires process to help each other. If a node is marked for delete, then a process trying to perform a different operation first needs to complete the deletion, before applying its own operation, otherwise the physical delete of the node may not take place, leaving the node forever in the stack. As the original algorithm avoids any marking, and simply tries to swing the Top pointer, we would like to maintain this property.

A field *popby* is initialised to $\perp$ when a node *nd* is created. Once the node is successfully insert to the stack by a Push operation, the inserting process tries to mark it by changing *popby* to Null using a *CAS*. Before a process tries to remove the node from the stack during a Pop routine, it first mark it as part of the stack by doing the same thing, helping the inserting process conclude the node is in the stack. This replace the logic delete of the node, as we only need to know the node was part of the stack if it is removed. After a successful *CAS* to remove *nd* from the stack, another *CAS* is used in order to try and set *popby* to the identifier of the process who performed the *CAS*. The use of *CAS* to change *popby* from $\perp$to Null, and from Null to an identifier guarantee that only the first process to perform each of these *CAS* will succeed. Note that before writing an identifier to *popby* a process must try and set it to Null, and thus it can not store two different identifiers along in any execution.

The correctness proof follows the same guidelines as of the proof for the Linked-List. If a Push operation did not introduce a new node *nd* into the stack, then no process but p is aware of *nd*. Thus, upon recovery the Search routine will not find *nd* in the stack, nor its *popby* field has been changed, and the Push-Roceover returns Fail. Otherwise, *nd* was successfully inserted to the stack. As discessed above, the only way to delete *nd* from the stack is by first changing its *popby* field to Null. Thus, upon recovery p will either find *nd* in the stack, using the Search routine, or that *popby* is different then $\perp$ in case it was deleted, and in both cases it returns **true**. For the Pop routine, if p tries to remove a node *nd* from the top of the stack and crash, then upon recovery it first check if *nd* is still in the stack using the Search routine. If it is so, then clearly *nd* was yet to delete, and it returns Fail. Otherwise, *nd* was deleted, either by p or by some other process. Only the first process of which to performs a *CAS*, writing its identifier to *popby* will return the value stored in *nd*, while the others return either $\perp$ (in the TryPop routine) or Fail (in the Pop-Recover routine).

Notice that both Push-Roceover and Pop-Recover are wait-free. Due to the structure of stack, no *next* pointer of any node in the stack is ever changed. Therefore, once a process reads Top at the beginning of its Recover routine, the chain of pointers from this Top to the last node in the stack is fixed for the rest of the execution, and thus traversing it using the Search routine is wait-free.

References

[1] Marcos Kawazoe Aguilera and Svend Frølund. 2003. Strict Linearizability and the Power of Aborting. In *Tech. Rep. HPL-2003-241*.

[2] Hagit Attiya, Ohad Ben-Baruch, and Hendler Danny. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 37th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*.

[3] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*.

[4] Naama Ben-David, Guy E. Blelloch, and Yuanhao Wei. 2018. Making Concurrent Algorithms Detectable. *CoRR* abs/1806.04780 (2018). arXiv:1806.04780 http://arxiv.org/abs/1806.04780

[5] Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. 2015. Robust Shared Objects for Non-Volatile Main Memory. In *19th International Conference on Principles of Distributed Systems, OPODIS*. 20:1–20:17.

[6] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 105–118.

[8] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient logging in non-volatile memory by exploiting coherency protocols. *PACMPL* 1, OOPSLA (2017), 67:1–67:24.

[9] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC*. 131–140.

[10] Wojciech M. Golab and Aditya Ramaraju. 2016. Recoverable Mutual Exclusion: [Extended Abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC*. 65–74.

[11] Rachid Guerraoui and Ron R. Levy. 2004. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *24th International Conference on Distributed Computing Systems (ICDCS)*. 400–407.

[12] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*. 300–314. https://doi.org/10.1007/3-540-45414-4_21

[13] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2010. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.* 70, 1 (2010), 1–12. https://doi.org/10.1016/j.jpdc.2009.08.011

[14] Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

[15] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[16] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC*. 313–327.

[17] Maged M Michael and Michael L Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 267–275.

[18] Shahar Timnat and Erez Petrank. 2014. A Practical Wait-free Simulation for Lock-free Data Structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 357–368.

[19] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *FAST*, Vol. 11. 61–75.

```
type Update {                    ▹ stored in one CAS word
    {CLEAN, DFlag, IFlag, MARK} state
    Flag *info
}
type Internal {                  ▹ subtype of Node
    Key ∪ {∞1, ∞2} key
    Update update
    Node *left, *right
}
type Leaf {                      ▹ subtype of Node
    Key ∪ {∞1, ∞2} key
}
type IInfo {                     ▹ subtype of Flag
    Internal *p, *newInternal
    Leaf *l
    boolean done
}
type DInfo {                     ▹ subtype of Flag
    Internal *gp, *p
    Leaf *l
    Update pupdate
    boolean done
}
▹ Initialization:
shared Internal *Root := pointer to new Internal node
    with key field ∞2, update field ⟨CLEAN, NULL⟩, and
    pointers to new Leaf nodes with keys ∞1 and
    ∞2, respectively, as left and right fields.
```

Figure 1. BST type definitions and initialization.

```
INSERT-RECOVER() {
    IInfo *op = RD[pid]
    if CP_p < ?? or op = ⊥ then
        Proceed from line ??
    test := op → p → update
    if test = ⟨IFlag, op⟩ then HELPINSERT(op)          ▹ Finish the insertion
    if op → done = TRUE then return TRUE
    else Proceed from line ??
}
DELETE-RECOVER() {
    DInfo *op = RD[pid]
    if CP_p < ?? or op = ⊥ then
        Proceed from line ??
    test := op → gp → update
    if test = ⟨DFlag, op⟩ then HELPDELETE(op)          ▹ Either finish deletion or unflag
    if op → done = TRUE then return TRUE
    else Proceed from line ??
}
```

Figure 2. RECOVER routines

```
SEARCH(Key k) : ⟨Internal*, Internal*, Leaf*, Update, Update⟩ {
    ▷ Used by INSERT, DELETE and FIND to traverse a branch of the BST; satisfies following postconditions:
    ▷ (1) l points to a Leaf node and p points to an Internal node
    ▷ (2) Either p → left has contained l (if k < p → key) or p → right has contained l (if k ≥ p → key)
    ▷ (3) p → update has contained pupdate
    ▷ (4) if l → key ≠ ∞_1, then the following three statements hold:
    ▷     (4a) gp points to an Internal node
    ▷     (4b) either gp → left has contained p (if k < gp → key) or gp → right has contained p (if k ≥ gp → key)
    ▷     (4c) gp → update has contained gpupdate
    Internal *gp, *p
    Node *l := Root
    Update gpupdate, pupdate                                        ▷ Each stores a copy of an update field

    while l points to an internal node {
        gp := p                                                     ▷ Remember parent of p
        p := l                                                      ▷ Remember parent of l
        gpupdate := pupdate                                         ▷ Remember update field of gp
        pupdate := p → update                                       ▷ Remember update field of p
        if k < l → key then l := p → left else l := p → right       ▷ Move down to appropriate child
    }
    return ⟨gp, p, l, pupdate, gpupdate⟩
}

FIND(Key k) : Leaf* {
    Leaf *l

    ⟨–, –, l, –, –⟩ := SEARCH(k)
    if l → key = k then return l
    else return NULL
}

INSERT(Key k) : boolean {
    Internal *p, *newInternal
    Leaf *l, *newSibling
    Leaf *new := pointer to a new Leaf node whose key field is k
    Update pupdate, result
    IInfo *op

    RD[pid] := ⊥
    CP_p := curPC()                                                 ▷ Set a check-point indicating IInfo structure was installed
    while TRUE {
        ⟨–, p, l, pupdate, –⟩ := SEARCH(k)
        if l → key = k then return FALSE                            ▷ Cannot insert duplicate key
        if pupdate.state ≠ CLEAN then HELP(pupdate)                 ▷ Help the other operation
        else {
            newSibling := pointer to a new Leaf whose key is l → key
            newInternal := pointer to a new Internal node with key field max(k, l → key),
                update field ⟨CLEAN, NULL⟩, and with two child fields equal to new and newSibling
                (the one with the smaller key is the left child)
            op := pointer to a new IInfo record containing ⟨p, l, newInternal, FALSE⟩
            RD[id] := op
            result := CAS(p → update, pupdate, ⟨IFlag, op⟩)         ▷ iflag CAS
            if result = pupdate then {                              ▷ The iflag CAS was successful
                HELPINSERT(op)                                      ▷ Finish the insertion
                return TRUE
            }
            else HELP(result)                 ▷ The iflag CAS failed; help the operation that caused failure
        }
    }
}

HELPINSERT(IInfo *op) {
    ▷ Precondition: op points to an IInfo record (i.e., it is not NULL)
    CAS-CHILD(op → p, op → l, op → newInternal)                     ▷ ichild CAS
    op → done := TRUE                                               ▷ announce the operation completed
    CAS(op → p → update, ⟨IFlag, op⟩, ⟨CLEAN, op⟩)                  ▷ iunflag CAS
}
```

Figure 3. Pseudocode for SEARCH, FIND and INSERT.

```
Delete(Key k) : boolean {
    Internal *gp, *p
    Leaf *l
    Update pupdate, gpupdate, result
    DInfo *op
    RD[pid] := ⊥
    CP_p := curPC()                                              ▷ Set a check-point indicating IInfo structure was installed
    while True {
        ⟨gp, p, l, pupdate, gpupdate⟩ := Search(k)
        if l → key ≠ k then return False                         ▷ Key k is not in the tree
        if gpupdate.state ≠ Clean then Help(gpupdate)
        else if pupdate.state ≠ Clean then Help(pupdate)
        else {                                                   ▷ Try to flag gp
            op := pointer to a new DInfo record containing ⟨gp, p, l, pupdate, False⟩
            RD[id] := op
            result := CAS(gp → update, gpupdate, ⟨DFlag, op⟩)    ▷ dflag CAS
            if result = gpupdate then {                          ▷ CAS successful
                if HelpDelete(op) then return True               ▷ Either finish deletion or unflag
            }
            else Help(result)            ▷ The dflag CAS failed; help the operation that caused the failure
        }
    }
}
HelpDelete(DInfo *op) : boolean {
    ▷ Precondition: op points to a DInfo record (i.e., it is not Null)
    Update result                                                ▷ Stores result of mark CAS
    result := CAS(op → p → update, op → pupdate, ⟨Mark, op⟩)     ▷ mark CAS
    if result = op → pupdate or result = ⟨Mark, op⟩ then {       ▷ op → p is successfully marked
        HelpMarked(op)                                           ▷ Complete the deletion
        return True                                              ▷ Tell Delete routine it is done
    }
    else {                                                       ▷ The mark CAS failed
        Help(result)                                             ▷ Help operation that caused failure
        CAS(op → gp → update, ⟨DFlag, op⟩, ⟨Clean, op⟩)          ▷ backtrack CAS
        return False                                             ▷ Tell Delete routine to try again
    }
}
HelpMarked(DInfo *op) {
    ▷ Precondition: op points to a DInfo record (i.e., it is not Null)
    Node *other
    ▷ Set other to point to the sibling of the node to which op → l points
    if op → p → right = op → l then other := op → p → left else other := op → p → right
    ▷ Splice the node to which op → p points out of the tree, replacing it by other
    CAS-Child(op → gp, op → p, other)                            ▷ dchild CAS
    op → done := True                                            ▷ announce the operation completed
    CAS(op → gp → update, ⟨DFlag, op⟩, ⟨Clean, op⟩)              ▷ dunflag CAS
}
Help(Update u) {                                                 ▷ General-purpose helping routine
    ▷ Precondition: u has been stored in the update field of some internal node
    if u.state = IFlag then HelpInsert(u.info)
    else if u.state = Mark then HelpMarked(u.info)
    else if u.state = DFlag then HelpDelete(u.info)
}
CAS-Child(Internal *parent, Node *old, Node *new) {
    ▷ Precondition: parent points to an Internal node and new points to a Node (i.e., neither is Null)
    ▷ This routine tries to change one of the child fields of the node that parent points to from old to new.
    if new → key < parent → key then
        CAS(parent → left, old, new)
    else
        CAS(parent → right, old, new)
}
```

Figure 4. Pseudocode for Delete and some auxiliary routines.

```
Type Node {
    T value
    int popby
    Node *next
}
Type CSInfo {                        ▷ subtype of Info
    Node *nd
    T result
}
Type ExInfo {                        ▷ subtype of Info
    {EMPTY, WAITING, BUSY} state
    T value, result
    ExInfo *partner, *slot
}
```

Figure 5. Elimination-Stack type definition

Algorithm 5: Recoverable Elimination-Stack: EXCHANGE routine.

ExInfo *default* - global static ExInfo object with state = EMPTY

Procedure T EXCHANGE (ExInfo **slot*, T *myitem*, long *timeout*)

```
long timeBound := getNanos() + timeout
ExInfo myop := new ExInfo(WAITING, myitem, ⊥, ⊥, slot)
RD[pid] := myop                                              // update Info structure
while true do
    if getNanos() > timeBound then
        return TIMEOUT
    yourop := slot
    switch yourop.state do
        case EMPTY
            myop.state := WAITING                            // attempt to replace default
            myop.partner := ⊥
            if slot.CAS(yourop, myop) then                   // try to collide
                while getNanos() < timeBound do
                    yourop := slot
                    if yourop ≠ myop then                    // a collision was done
                        if youop.parnter = myop then         // yourop collide with myop
                            SWITCHPAIR(myop, yourop)
                            slot.CAS(yourop, default)        // release slot
                        return myop.result
                // time limit reached and no process collide with me
                if slot.CAS(myop, default) then              // try to release slot
                    return TIMEOUT
                else                                         // some process show up
                    yourop := slot
                    if yourop.partner = myop then
                        SWITCHPAIR(myop, yourop)             // complete the collision
                        slot.CAS(yourop, default)            // release slot
                    return myop.result
            break
        case WAITING                                         // some process is waiting in slot
            myop.partner := yourop                           // attempt to replace yourop
            myop.state := BUSY
            if slot.CAS(yourop, myop) then                   // try to collide
                SWITCHPAIR(myop, yourop)                     // complete the collision
                slot.CAS(myop, default)                      // release slot
                return myop.result
            break
        case BUSY                                            // a collision in progress
            SWITCHPAIR(yourop, yourop.parnter)               // help to complete the collision
            slot.CAS(yourop, default)                        // release slot
            break
```

Algorithm 6: Recoverable Elimination-Stack: Elimination Array routines.

Procedure void SwitchPair(ExInfo *first*, ExInfo *second*)

```
/* exchange the valus of the two operations */
first.result := second.value
second.result := first.value
```

Procedure T Visit (T *value*, int *range*, long *duration*)

```
/* invoke Exchange on a random entery in the collision array */
int cell := randomNumber(range)
return Exchange(exchanger[cell], value, duration)
```

Procedure T Exchange-Recover (ExInfo **myop*)

```
ExInfo *slot := myop.slot                              // slot to recover
if myop.result ≠ ⊥ then                                // If operation response was persisted
    return myop.result
if myop.state = Waiting then
    /* crash while trying to exchange defualt, or waiting for a collision */
    yourop := slot
    if yourop = myop then                              // still waiting for a collision
        if ¬slot.CAS(myop, default) then               // try to release slot
            yourop := slot                             // some process show up; complete collision
            if yourop.partner = myop then
                SwitchPair(myop, yourop)               // complete the collision
                slot.CAS(yourop, default)              // release slot
    else if yourop.partner = myop then                 // yourop collide with myop
        SwitchPair(myop, yourop)                       // complete the collision
        slot.CAS(yourop, default)                      // release slot
if myop.state = Busy then
    /* crash while trying to collide with myop.partner */
    yourop := slot
    if yourop = myop then                              // collide was successful and in progress
        SwitchPair(myop, myop.partner)                 // complete the collision
        slot.CAS(myop, default)                        // release slot
return myop.result
```

Algorithm 7: Recoverable Elimination-Stack: PUSH routines.

Procedure boolean TRYPUSH (Node *nd)

```
    /* attempt to perform PUSH to the central stack                                        */
159 Node *oldtop := Top
160 nd.next := oldtop
161 RD[pid] := data                                  // update Info structure for this operation
162 if Top.CAS(oldtop, nd) then                          // try to declare nd as the new Head
163     nd.popby.CAS(⊥, NULL)                                  // announce nd is in the stack
164     data.result := true                                               // Persist response
165     return true
166 return false
```

Procedure boolean PUSH (T myitem)

```
167 Node *nd = new Node (myitem, ⊥, ⊥)
168 CSInfo *data := new CSInfo (nd, ⊥)
169 RD[pid] := data                                 // Install Info structure for this operation
170 CP_p := curPC()              // Set a check-point indicating Info structure was installed
171 while true do
172     if TRYPUSH(nd) then                               // if central stack PUSH is successful
173         return true
174     range := CalculateRange()                          // get parameters for collision array
175     duration := CalculateDuration()
176     othervalue := VISIT(myitem, range, duration)                           // try to collide
177     if othervalue = NULL then                     // successfuly collide with POP operation
178         RecordSuccess ()
179         return true
180     else if othervalue = TIMEOUT then                                   // failed to collide
181         RecordFailure ()
```

Procedure boolean PUSH-ROCEOVER ()

```
182 info data := RD[pid]                                                // read recovery data
183 T result := ⊥
184 if CP_p < 170 then
185     Proceed from line 167             // Failed before installing info structure, re-execute
186 if data of type ExInfo then                          // crash while accessing collision array
187     if EXCHANGE-RECOVER(data) = NULL then                              // successful collision
188         result := true
189 else                                                   // crash while accessing central stack
190     Node *nd := RD[pid].nd
191     if RD[pid].result ≠ ⊥ then                         // If operation response was persisted
192         result := RD[pid].result
193     else if SEARCH(nd) || nd.popby ≠ ⊥ then       // nd in the stack, or was announced as such
194         nd.popby.CAS(⊥, NULL)                                  // announce nd is in the stack
195         result := true
196 if result ≠ ⊥ then                                               // operation was completed
197     RD[pid].result := result                                /* persist response and return */
198     return result
199 else                                                         // operation was not completed
200     Proceed from line 167                                                       // re-execute
```

Procedure boolean SEARCH (Node *nd)

```
    /* search for node nd in the stack                                                       */
201 Node *iter := Top
202 while iter ≠ ⊥ do
203     if iter = nd then
204         return true
205     iter := iter.next
206 return false
```

Algorithm 8: Recoverable Elimination-Stack: Pop routines.

Procedure T TryPop()

```
207 Node *oldtop := Top
208 Node *newtop
209 data.nd := oldtop
210 RD[pid] := data                              // update Info structure for this operation
211 if oldtop = ⊥ then                                                  // stack is empty
212 |   data.result := Empty                                           // Persist response
213 |   return Empty
214 newtop := oldtop.next
215 oldtop.popby.CAS(⊥,Null)                                // announce oldtop is in the stack
216 if Top.CAS(oldtop,newtop) then              // try to pop oldtop by changing Top to newtop
217 |   if newtop.popby.CAS(Null,pid) then                 // try to announce yourself as winner
218 |   |   data.result := oldtop.value                                 // Persist response
219 |   |   return oldtop.value
220 else
221 |   return ⊥
```

Procedure T Pop ()

```
222 Node *result
223 CSInfo *data := new CSInfo (Top,⊥)
224 RD[pid] := data                             // Install Info structure for this operation
225 CP_p := curPC()             // Set a check-point indicating Info structure was installed
226 while true do
227 |   result := TryPop()                                // attempt to pop from central stack
228 |   if result ≠ ⊥ then                               // if central stach Pop is successful
229 |   |   return result
230 |   range := CalculateRange()                      // get parameters for collision array
231 |   duration := CalculateDuration()
232 |   othervalue := Visit(Null,range,duration)                             // try to collide
233 |   if othervalue = Timeout then                                      // failed to collide
234 |   |   RecordFailure ()
235 |   else if othervalue ≠ Null then              // successfuly collide with Push operation
236 |   |   RecordSuccess ()
237 |   |   return othervalue
```

Procedure T Pop-Recover()

```
238 info data := RD[pid]                                                // read recovery data
239 T result := ⊥
240 if CP_p < 225 then
241 |   Proceed from line 222           // Failed before installing info structure, re-execute
242 if data of type ExInfo then                           // crash while accessing collision array
243 |   temp := Exchange-Recover(data)
244 |   if temp ≠ Null && temp ≠ ⊥ then                                   // successful collision
245 |   |   result := temp
246 else                                                    // crash while accessing central stack
247 |   Node *nd := RD[pid].nd
248 |   if RD[pid].result ≠ ⊥ then                           // If operation response was persisted
249 |   |   result := RD[pid].result
250 |   else if nd = ⊥ then                                             // pop from an empty stack
251 |   |   result := Empty
252 |   else if ¬Search(nd) then                                   // nd was removed from the stack
253 |   |   nd.popby.CAS(Null,pid)                           // try to announce yourself as the winner
254 |   |   if nd.popby = pid then                                         // if you are the winner
255 |   |   |   result := nd.value
256 if result ≠ ⊥ then                                                    // operation was completed
257 |   RD[pid].result := result                                    /* persist response and return */
258 |   return result
259 else                                                              // operation was not completed
```