1 Linked-List

Harris Linked-List uses an Atomic-Markable-Reference object, in which the next field of a Node, in addition to a reference to the next node in the list, is also marked or unmarked. The two fields can be update atomically, either together or individually. This can be done by using the most-significant-bit of next for the marking. For simplicity, we assume reading next returns the reference only, while get_data() function is used to get (atomically) both the reference and mark bit. Moreover, whenever performing CAS on node.next, both reference and mark state should be mention. For ease of presentation, we assume a List is initialized with head and tail, containing keys $-\infty, \infty$ respectively. We allow no insert or delete of these keys.

A brief description of the original implementation and its linearization is as follows. The Lookup procedure is used by Insert and Delete in order to find the node with the lowest key greater or equal to the input key, and its predecessor on the list, while physically removing any marked node on its way. To insert a key α, a process first finds the right location for α using the Lookup procedure, and then tries to set pred.next to point to a new node containing α by performing CAS. To delete a key α, a process search for it using the Lookup procedure, and then tries to logically delete it by marking the next field using CAS. In case the marking was successful, the process also tries to physically remove the node. To find a key α, a process simply looks for a node in the list with key α which is unmarked.

The linearization point for the original implementation are as follows:

Insert: At the point of a successful CAS

Delete: At the point of a successful CAS for marking the node (logical delete)

Find: At the point where the procedure return, that is, at the read of either curr.key or curr.next.

Following the given linearization points (omitting proof...), insert and delete operation are linearized at the point where they affect the system. That is, if an insert operation performed a successful CAS, then all process will see the new node starting from this point, and if a node was logically delete, then all processes treat it as if it was removed. Therefore, once a process p recovers following a crash, the list data structure is consistent - if p has a pending operation, either the operation already had a linearization point which affected all other processes, or it did not affect the data structure at all, nor will in any future run.

However, even though the list data structure is consistent, the response of the pending operation is lost. Consider for example a scenario in which process p performs $Delete(\alpha)$ and crash right after applying a successful CAS to mark a node. Upon recovery, p may be able to decide α was removed, as the node is marked. Nevertheless, even if no other process takes steps, p is not able to determine whether it is the process to successfully delete α, or that it was done by some other process, and therefore it does not able to determine the right response. Moreover, in case the node was physically removed, p is not able to determine whether α has been deleted at all, as it is no longer part of the list.

1.0.1 Linked-List Recoverable Version

To solve the problems mention above, we present a modification for the algorithm such that in case of a process crash, upon recovery it is able to complete its last pending operation if needed, and also return the response value in such case. The algorithm presented in figure 1. Blue lines represents changes comparing to the original algorithm.

Each node is equipped with a new field named deleter. This field is used to determine which process is the one to delete the node. After the node was successfully marked (logical delete), process p tries to announce itself as the one to delete the node by writing its id to deleter using CAS. This way, if a process crash during a delete, it can use deleter in order to determine the response value. We assume deleter is initialised to null when creating a new node.

Each process p has a designated location in the memory, Backup[p]. Before trying to apply an operation, p writes to Backup[p] the entire data needed to complete the operation. Upon recovery, p can read Backup[p], and based on it to complete its pending operation, in case there is such. Formally, Backup[p] contains a pointer to a structure containing all the relevant data. For simplicity, process p creates new such structure for each of its operations, although a more efficient way will be to use two such structures in an alternating way.

Algorithm 1: ⟨Node, Node⟩ SEARCH (T *key*)

```
Data:   Node *pred, *curr, *succ
        boolean mbit
retry: while true do
    pred := head
    curr := pred.next
    while true do
        ⟨succ, mbit⟩ := curr.next.get_data()
        if mbit then                                                   // succ was logically deleted
            if pred.next.CAS(unmarked curr,unmarked succ) = false then // help physical delete
                go to retry                                            // help failed
            curr := succ                                               // help succeed
        else
            if curr.key ≥ key then              // curr is the first unmarked node with key ≥ key
                return ⟨pred, curr⟩
            pred := curr                                               // advance pred and curr
            curr := succ
    end
end
```

Algorithm 5: boolean RECOVER ()

```
Data:    Node *nd := Backup[pid].nd
if Backup[pid].result ≠ ⊥ then                              // operation was completed
    return Backup[pid].result
if Backup[pid].optype = INSERT then
    ⟨pred, curr⟩ := SEARCH(nd.key)                           // search for nd in the list
    if curr = nd || nd.next is marked then                   // nd is in the list or marked
        Backup[pid] := true
        return true
    return FAIL
if Backup[pid].optype = DELETE then
    if nd ≠ ⊥ && nd.next is marked then                      // nd was logically deleted
        nd.deleter.CAS(⊥, pid)                               // try to complete the deletation
        if nd.deleter = pid then                             // you are the deleter
            Backup[pid].result := true
            return true
    return FAIL
```

Shared variables: Node **head*

Type Info {
    {INSERT, DELETE} *optype*
    Node **nd*
    boolean *result*
}

Code for process p:

Algorithm 2: boolean INSERT (T *key*)

```
Data:   Node *pred, *curr
        Node newnd := new Node (key)
Backup[pid] := new Info (INSERT, newnd, ⊥)
while true do
    ⟨pred, curr⟩ := SEARCH(key)                 // search for the right location for insertion
    if curr.key = key then                       // key is already in the list
        Backup[pid].result := false
        return false
    else
        newnd.next := unmarked curr
        if pred.next.CAS (unmarked curr, unmarked newnd) then        // try to add newnd
            Backup[pid].result := true
            return true
end
```

Algorithm 3: boolean DELETE (T *key*)

```
Data:   Node *pred, *curr, *succ
Backup[pid] := new Info (DELETE, ⊥, ⊥)
⟨pred, curr⟩ := SEARCH(key)                       // search for key in the list
if curr.key ≠ key then                            // key is not in the list
    Backup[pid].result := false
    return false
else
    Backup[pid].nd := curr
    while curr.next is unmarked do                // repeatedly attempt logical delete
        succ := curr.next
        curr.next.CAS (unmarked succ, marked succ)
    end
    succ := curr.next
    pred.next.CAS (unmarked curr, unmarked succ)   // physical delete attempt
    res := curr.deleter.CAS(⊥, pid)                // try to announce yourself as deleter
    Backup[pid].result := res
    return res
```

Algorithm 4: boolean FIND (T *key*)

```
Data:   Node *curr := head
while curr.key < key do        // search for the first node with key greater or equal to key
    curr = curr.next
end
return (curr.key = key && curr.next is unmarked)
```

Figure 1: Recoverable Non-Blocking Linked-List

Correctness Argument

In the following, we give an high-level proof for the correctness of the algorithm.

First, notice that quitting the Lookup procedure at any point, or repeating it, can not violet the list consistency. The Lookup procedure simply traverse the list, while trying to physically delete marked nodes. Once curr.next is marked, a single process can perform the physical delete. This follows from the fact that at any point there is a single node in the list which points to curr. Once curr is physically delete, no node in the list points to curr, and thus any CAS operation with curr as the first parameter will fail. This observation relays on the fact that any new allocated node has a different address then curr. As a result, repeating the attempt to physically delete a node does not affect the list.

Assume a process p performs an $insert(key)$ operation. First, p writes to Backup[p], updating it is about to perform an Insert. If a process p does not crash, then, as in the original algorithm, it repeatedly tries to find the right location for the new node, and insert it by performing a CAS changing *pred.next* to point to newnd. In addition, it is clear from the code that a crash after updating $Backup[p].result$ is after the operation had its linearization point, and the Recover procedure will return the right response. Therefore, we need to consider a crash before an update to $Backup[p].result$. There are two scenarios to consider.

Assume p crash without performing a successful CAS in line 27. p is the only process to have a reference to newnd, and it is yet to update any node with this reference, and thus no node points to newnd. As a result, the operation did not affects any other process, nor it will be in the future. Hence, considering the operation as not having a linearization point does not violate the list consistency. Indeed, since no node points to newnd, upon recovery p will see that newnd is not in the list and also not marked, and thus will return FAIL. Notice this argument holds whether key is already in the tree, or not, as the operation in both cases did not affect the system.

Assume now p crash after performing a successful CAS in line 27. In such case, newnd is part of the list, as pred.next points to it. Also notice we did not delete any other node, since pred.next pointed to curr, and after the CAS it points to newnd which points to curr. As a result, when p executes the Recover procedure, either it will see newnd in the list, or that it is no longer part of the list, and it must be some other process deleted it, and hence newnd.next is marked. In any case, p will return true as required. The above argument relies on the fact a marked node can not be unmarked, and that an Insert and Delete can not mistakenly remove nodes from the list. We have claimed it for Insert, and we will prove the same holds for Delete. Therefore, if a node is no longer in the list, it must be marked.

Assume a process p performs a $delete(key)$ operation. First, p writes to Backup[p], updating it is about to perform a Delete. As before, a crash after writing to Backup[p].result will return the right response. Also, a crush before updating any of Backup[p].result or Backup[p].nd implies p is yet to try and mark any node, and thus the operation did not affect the system so far, nor it will be in the future. Therefore, we can consider the operation as not having a linearization point (even in case key is not the list), and indeed, the Recover procedure returns FAIL in such case.

Assume thus p writes to Backup[p].nd. It follows that p completed the lookup procedure and finds a node curr storing key. The lookup procedure guarantees there is a point in time (of the procedure execution) where curr is in the list and curr.next is not marked. If p crash and recovers, and observe that curr is unmarked, then in returns FAIL. Since a marked node can not be unmarked, as there is no CAS changing a marked node, it follows that p did not marked curr. Therefore, the operation did not affects any process, nor it will be, and we consider it as having no linearization point. Otherwise, the Recover function observe curr as marked, and we can conclude the marking point of curr is along the delete operation. We now prove we can linearize the operation, according to its response.

Let q be the process to mark curr. Since once curr.next is marked it will never be changed, the reference of curr.next is fixed to succ (of q at the point of the marking). This also implies q is unique and well defined, and any future CAS on curr.next will fail. As a result, any process leaving the while loop in line 39 reads the same value in line 43, which is this succ. The attempt to physically delete curr in line 44 will succeed only if pred.next points to curr, and as we said, curr point to succ, and any other attempt will fail. Thus, if this attempt succeed, it deletes only curr, and can not delete additional nodes.

In line 45 process p tries to writes its id to curr.deleter. As it is initialised to null, only the first process to perform this CAS will succeed. Also, any p must go through line 45 in order to complete its operation, as the Recover procedure redirect the process to this line. Therefore, if there is a process to complete its delete operation while observing curr.next is marked, there must be a CAS to curr.deleter. Let q' be the first process to perform this CAS. As proved above, q' tries to delete curr, and the point in time where curr is marked must be contained in its operation interval. Moreover, q' is the only process to write to curr.deleter, and the first one to do so, thus q' is the only process to obtain true when testing (curr.deleter = q') in line 46 (and thus to also return true), while any other process will obtain false. We linearize the operation of q' at the point of the marking, and any other attempt to delete curr is linearized after it (in an arbitrary order).

A corollary of the analysis is that processes trying to delete the same node curr "helps" each other, in the sense that they all keep trying to mark curr. However, the marking process is not necessarily the one to return true. Also, in the original algorithm, if a process fails to mark a node, it starts the delete operation from the beginning. In our implementation, process can keep trying to mark the node without the need to perform a lookup again after each failed CAS. We guarantee that once curr is marked, exactly one process will return true, while the rest can consider curr as being deleted (in the course of their delete execution), and thus there is a point along their execution is which key is not in the tree, and they can return false.

2 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 sub-routines, 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.

2.1 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 81, the only way to overwrite it by a different process q, is by a **CAS** in line 104 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 81, 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 104, 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.

2.2 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.

Type Node {
    T *value*
    int *popby*
    Node $*next$
}

Type PushInfo { $\triangleright$ subtype of Info
    Node $*pushnd$
}

Type PopInfo { $\triangleright$ subtype of Info
    Node $*popnd$
}

Type ExInfo { $\triangleright$ subtype of Info
    {EMPTY, WAITING, BUSY} *state*
    T $value, result$
    ExInfo $*partner, *slot$
}

Figure 2: Type definition

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

Algorithm 6: T EXCHANGE (ExInfo *slot, T myitem, long timeout)

```
long timeBound := getNanos() + timeout
ExInfo myop := new ExInfo(WAITING, myitem, ⊥, ⊥, slot)
Announce[pid] := myop
while true do
    if getNanos() > timeBound then
        myop.result := TIMEOUT                                  // time limit reached
        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
                end
                // time limit reached and no process collide with me
                if slot.CAS(myop, default) then                 // try to release slot
                    myop.result := TIMEOUT
                    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
            end
            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
    endsw
end
```

Algorithm 7: void SwitchPair(ExInfo *first*, ExInfo *second*)

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

Algorithm 8: 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)
```

Algorithm 9: T Exchange-Recover ()

```
ExInfo *myop := Announce[pid]                   // read your last operation record
ExInfo *slot := myop.slot                       // and the slot on which it acts
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
            return Fail
        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
    else if yourop.partner = myop then          // yourop collide with myop
        SwitchPair(myop, yourop)                // complete the collision
        slot.CAS(yourop, default)               // release slot
        return myop.result
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
if myop.result ≠ ⊥ then
    return myop.result                          // collide was successfuly completed
else
    return Fail
```

Figure 3: Elimination Array routines

Algorithm 10: boolean TryPush (Node *nd)

```
/* attempt to perform Push to the central stack                          */
Node *oldtop := Top
nd.next := oldtop
if Top.CAS(oldtop, nd) then                  // try to declare nd as the new Head
    nd.popby.CAS(⊥, Null)                           // announce nd is in the stack
    return true
return false
```

Algorithm 11: boolean Push (T myitem)

```
Node *nd = new Node (myitem)
nd.popby := ⊥
PushInfo data := new PushInfo (nd)
while true do
    Announce[pid] := data                     // declare - trying to push node nd
    if TryPush(nd) then                    // if central stack Push is successful
        return true
    range := CalculateRange()              // get parameters for collision array
    duration := CalculateDuration()
    othervalue := Visit(myitem, range, duration)                // try to collide
    if othervalue = Null then          // successfuly collide with Pop operation
        RecordSuccess ()
        return true
    else if othervalue = Timeout then                        // failed to collide
        RecordFailure ()
end
```

Algorithm 12: boolean Push-Roceover ()

```
Node *nd := Announce[pid].pushnd
if nd.popby ≠ ⊥ then                        // nd was announced to be in the stack
    return true
if Search(nd) || nd.popby ≠ ⊥ then  // nd in the stack, or was announced as such
    nd.popby.CAS(⊥, Null)                           // announce nd is in the stack
    return true
return Fail
```

Algorithm 13: boolean Search (Node *nd)

```
/* search for node nd in the stack                                       */
Node *iter := Top
while iter ≠ ⊥ do
    if iter = nd then
        return true
    iter := iter.next
end
return false
```

Figure 4: Push routine

Algorithm 14: T TRYPOP()

```
Node *oldtop := Top
Node *newtop
Announce[pid].popnd := oldnop          // declare - trying to pop node oldtop
if oldtop = ⊥ then                     // stack is empty
    return EMPTY
newtop := oldtop.next
oldtop.popby.CAS(⊥, NULL)              // announce oldtop is in the stack
if Top.CAS(oldtop, newtop) then        // try to pop oldtop by changing Top to newtop
    if newtop.popby.CAS(NULL, pid) then   // try to announce yourself as winner
        return oldtop.value
else
    return ⊥
```

Algorithm 15: T POP ()

```
Node *result
PopInfo data := new PopInfo (Top)
while true do
    Announce[pid] := data                 // declare - trying to perform POP
    result := TRYPOP()                    // attempt to pop from central stack
    if result ≠ ⊥ then                    // if central stach POP is successful
        return result
    range := CalculateRange()             // get parameters for collision array
    duration := CalculateDuration()
    othervalue := VISIT(NULL, range, duration)   // try to collide
    if othervalue = TIMEOUT then          // failed to collide
        RecordFailure ()
    else if othervalue ≠ NULL then        // successfuly collide with PUSH operation
        RecordSuccess ()
        return othervalue
end
```

Algorithm 16: T POP-RECOVER()

```
Node *nd := Announce[pid].popnd        // crash while trying to pop node nd
if nd = ⊥ then                         // pop from an empty stack
    return EMPTY
if SEARCH(nd) then                     // nd was not removed from the stack
    return FAIL
nd.popby.CAS(NULL, pid)                // nd was removed.  Try to complete the operation
if nd.popby = pid then                 // you are the process to win the pop of nd
    return nd.value
return FAIL
```

Figure 5: POP routine