1 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 101 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 an Op_1 = Insert(k_1) operation. q performs a successful CAS in line 101 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 $\perp$, 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 1. 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 2. The intuition for it is drown 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.

1.0.1 Correctness

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 101 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 101 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 128 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 139, 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 128 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 154, or in line 145 of the HelpDelete routine. As mention before, the first CAS in line 138 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 $p \to update$, and hence no HelpMarked(op) is ever invoked. As a result, there is no write to $op \to done$. In such case, the Delete operation did not took affect, nor will be, and indeed q will return Fail.

```
Find(Key k) : Leaf* {
    Internal *gp, *p
    Leaf *l
    Update pupdate, gpupdate

    while (True) {
        ⟨gp, p, l, pupdate, gpupdate⟩ := Search(k)
        if gpupdate.state ≠ Clean then Help(gpupdate)
        else if pupdate.state ≠ Clean then Help(pupdate)
        else if gp → update = gpupdate and p → update = pupdate then {
            if l → key = k then return l
            else return ⊥
        }
    }
}
```

Figure 1: Solution 1: R-linearizable Find routine

```
Find(Key k) : Leaf* {
    Internal *gp, *p
    Leaf *l
    Update pupdate, gpupdate

    ⟨gp, p, l, pupdate, gpupdate⟩ := Search(k)
    if l → key ≠ k then {
        if (pupdate.state = IFlag and pupdate.info attempt to add key k) then
            return leaf with key k from pupdate.info
        else return ⊥
    }
    if (pupdate.state = Mark and pupdate.info → l → key = k) then return ⊥
    if (gpupdate.state = DFlag and gpupdate.info → l → key = k) then {
        op := gpupdate.info
        result := CAS(op → p → update, op → pupdate, ⟨Mark, op⟩)                ▷ mark CAS
        if (result = op → pupdate or result = ⟨Mark, op⟩) then return ⊥        ▷ op → p is successfully marked
    }
    return l
}
```

Figure 2: Solution 2: R-linearizable Find routine

```
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, ⊥⟩, and
    pointers to new Leaf nodes with keys ∞1 and
    ∞2, respectively, as left and right fields.
```

Figure 3: Type definitions and initialization.

```
Recover() {
    Flag *op = Announce[id]

    if op of type IInfo then {
        result := op → p → update                       ▷ Check flag
        if result = ⟨IFlag, op⟩ then HelpInsert(op)      ▷ Finish the insertion
    }
    if op of type DInfo then {
        result := op → gp → update                      ▷ Check flag
        if result = ⟨DFlag, op⟩ then HelpDelete(op)      ▷ Either finish deletion or unflag
    }
    if op → done = True then return True
    else return Fail
}
```

Figure 4: Recover routine

```
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 ⊥
}

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

    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, ⊥⟩, 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⟩
            Announce[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 ⊥)
    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 5: Pseudocode for Search, Find and Insert.

```
Delete(Key k) : boolean {
    Internal *gp, *p
    Leaf *l
    Update pupdate, gpupdate, result
    DInfo *op

    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⟩
            Announce[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 ⊥)
    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 ⊥)
    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 ⊥)
    ▷ 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 6: Pseudocode for Delete and some auxiliary routines.