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) and assume q performs a successful CAS in line 1 and fails after completing the HelpInsert routine. In this case, the Insert operation took effect, that is, the new key appears as a leaf of the tree, and any Find(k) operation will return it. However, even though the operation was already linearized at the time of the crash, upon recovery process q is unaware of it. Moreover, looking for the new leaf in the tree is not healpfull, since it might be k has been removed from the tree after the crush.

Moreover, 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 $Op_1 = \text{Insert}(k_1)$ operation. After a successful CAS in line 1 the process crush. After recovering, q invoke an $Op_2 = \text{Insert}(k_2)$ 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 inserting 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, follows by an Find(k_2), which returns the leaf of k_2. The Insert(k_1) operation will be completed later by any Insert or Delete 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 grandparent or parent that flagged, then it might be that an insert or delete of k is currently in a progress, thus we first help these 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 already 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.

```
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-linearizability Find routine

The correctness of the two 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 invokes 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 crush is not addressed. In general, the critical points in the code for recovery are the CAS primitives, as a crush 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, the expend the helping mechanism, so that the helping process needs also to update the info structure in case of a success. This is done by adding a boolean field to the Flag structure. This way, if a process crush along an operation *Op*, upon recovery it can check whether the operation was completed by some different process.

Before a process attempt to perform an operation, as it creates the Flagstructure *op* describing the operation and its affect on the data structure, the process stores *op* in a designated location (according to its id). Upon recovery, the process reads this location, and if the operation is not complete, then it retries to perform it, starting from the point of the first flagging (the first CAS). Otherwise, the operation was completed, and the response value is already known. Notice that there is a scenario in which process *q* recovers and observes an operation *Op* as not being complete, 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.

The supplied implementation does not specify what happens if a process crush outside of a BST 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-linearizability 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 complete
}
type DInfo {                  ▷ subtype of Flag
    Internal *gp, *p
    Leaf *l
    Update pupdate
    boolean complete
}
▷ 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
                if op ← complete = True then return True
                else go to line 1
        if op of type DInfo then
                if op ← complete = True then return True
                else go to line 1
}
```

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 or result = ⟨IFlag, op⟩ 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
        }
        if op → complete = True then
            return True
    }
}

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
    CAS(op → p → update, ⟨IFlag, op⟩, ⟨Clean, op⟩)              ▷ iunflag CAS
    op → complete := True                                       ▷ mark the operation as completed
}
```

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 or result = ⟨DFlag, op⟩ 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
        }
        if op → complete = TRUE then
            return TRUE
    }
}

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
    CAS(op → gp → update, ⟨DFlag, op⟩, ⟨CLEAN, op⟩)                                ▷ dunflag CAS
    op → complete := TRUE                                                         ▷ mark the operation as completed
}

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.