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 96 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 96 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 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 crash 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 Flag structure op describing the operation and its affect on the data structure, the process stores op in a designated location (for simplicity, we use an array). Upon recovery, the process reads this location, and if the operation is not completed yet, 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 completed, 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.

Notice that 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].done is updated only if the routine made changes to the BST.

1.0.1 Correctness

In this subsection 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 [....]. Moreover, 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. Thus, if we consider an execution and prove it is indistinguishable to a process q from some execution of the original algorithm in which more processes can participate, then it has to return the same response on both. The proof relies on several key arguments given below.

1. It is easy to verify the post-conditions of the SEARCH routine still holds, as they follow directly from the code. Also, since no changes are made to the SEARCH routine, it 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.

2. 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 deleted, or replace with a new copy when an insert operation takes affect. Otherwise, nd_2 is an internal node, and as such, it can not be replaced by an insert operation (which only allows to replace a leaf), and therefore it can only be removed from the tree.

3. The field update of a node *nd* can have any value only once along the execution. This follows from the fact that any attempt to perform an operation creates new record in the memory. If $nd \leftarrow update$ is marked, it can be unmarked or change. 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 $nd \leftarrow update$ 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. As a corollary, if a process successfully flag or mark a node, there was no change in the node since the last time it read the node update field.

Assume process q performs an INSERT(k) operation. As argued in argument 1, a crash before writing to Announce[q] implies no changes has been made to the BST. Assume thus q executes line 95, i.e., q stored in Announce[q] a pointer to an IInfo record containing all the data needed for the current attempt to complete the INSERT routine. Since q is in the middle of a while loop, it is enough to prove that if q crash before the next time it writes to Announce[q], if there is such write, upon recovery it will either complete its operation with the right response, or will continue to the next write to Announce[q] without having any affect on the BST. Hence, the same argument can be applied once q writes to Announce[q] again.

Assume q performs a successful CAS in line 96. Then, a reference to the IInfo *op* is stored in $p \leftarrow update$, which is also flagged. Following argument 3, p was not changed since the SEARCH routine read it, and it still points to l. Starting from this point, no changes can be made to p, except for the change point to by *op*, as the node is flagged. Now, only the first process

Relying on the correctness of the original algorithm, no matter how many times HELPINSERT(op) will be executed, the change will occur only once. This follows from the fact that many process can observe op, and will try to complete it in the future. The core for this argument is that a node never point twice to the same node.

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

    while (True) {
        $\langle gp, p, l, pupdate, gpupdate \rangle :=$ Search(k)
        if $gpupdate.state \neq$ Clean then Help($gpupdate$)
        else if $pupdate.state \neq$ Clean then Help($pupdate$)
        else if $gp \leftarrow update = gpupdate$ and $p \leftarrow update = pupdate$ then {
            if $l \rightarrow key = k$ then return l
            else return $\perp$
        }
    }
}

Figure 1: Solution 1: R-linearizable Find routine

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

    $\langle gp, p, l, pupdate, gpupdate \rangle :=$ Search(k)
    if $l \rightarrow key \neq k$ then {
        if ($pupdate.state =$ IFlag and $pupdate.info$ attempt to add key k) then
            return leaf with key k from $pupdate.info$
        else return $\perp$
    }
    if ($pupdate.state =$ Mark and $pupdate.info \leftarrow l \leftarrow key = k$) then return $\perp$
    if ($gpupdate.state =$ DFlag and $gpupdate.info \leftarrow l \leftarrow key = k$) then {
        $op := gpupdate.info$
        $result :=$ CAS($op \rightarrow p \rightarrow update, op \rightarrow pupdate, \langle \text{Mark}, op \rangle$) ▷ **mark** CAS
        if ($result = op \rightarrow pupdate$ or $result = \langle \text{Mark}, op \rangle$) then return $\perp$ ▷ $op \rightarrow p$ 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
      {CLEAN,DFlag,IFlag,MARK, TRUE} done
}
type DInfo {                     ▷ subtype of Flag
      Internal *gp, *p
      Leaf *l
      Update pupdate
      {CLEAN,DFlag,IFlag,MARK, TRUE} 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 ← done = TRUE then return TRUE
      if op of type IInfo then
            go to line 96
      if op of type DInfo then
            go to line 126
}
```

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, Clean⟩
            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 → done = True then
            return True
    }
}

HelpInsert(IInfo *op) {
    ▷ Precondition: op points to an IInfo record (i.e., it is not ⊥)
    op → done := IFlag                                ▷ announce the iflag CAS was successful
    CAS-Child(op → p, op → l, op → newInternal)       ▷ ichild CAS
    CAS(op → p → update, ⟨IFlag, op⟩, ⟨Clean, op⟩)    ▷ iunflag CAS
    op → done := True                                 ▷ announce the operation is 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, Clean⟩
            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 → done = 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 → done := 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.