1 Linked-List

The original algorithm by Harris is presented in Figure 1. Harris approach 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 node.next returns the reference, while a query can be used to identify if it is marked. Therefore, whenever writing to node.next, or performing CAS, both the reference and marking state should be mention. For ease of presentation, we assume a List is initialised with head and tail, containing keys $-\infty, \infty$ respectively. We allow no insert or delete of these keys.

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 looks for it using the Lookup procedure, and then tries to logically remove it by marking curr.next 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.

Procedure Lookup(int key)

```
Data: Node* pred, curr, succ
retry: while true do
    pred = head
    curr = head.next
    while true do
        succ = curr.next
        if curr.next is marked then
            if pred.next.CAS (unmarked curr,unmarked succ) == false then
                go to retry
            end
            curr = succ
        else
            if curr.key ≥ key then
                return ⟨pred,curr⟩
            end
            pred = curr
            curr = succ
        end
    end
end
```

Shared variables: Node* head

```
Procedure Insert(int key)
  Data: Node* pred, curr
        Node node = new Node (key)
  while true do
  |  ⟨pred, curr⟩ = lookup(key)
  |  if curr.key == key then
  |  |  return false
  |  else
  |  |  node.next = unmarked curr
  |  |  if pred.next.CAS (unmarked curr, unmarked node) then
  |  |  |  return true
  |  |  end
  |  end
  end
```

```
Procedure Delete(int key)
  Data: Node* pred, curr, succ
  while true do
  |  ⟨pred, curr⟩ = lookup(key)
  |  if curr.key != key then
  |  |  return false
  |  else
  |  |  succ = curr.next
  |  |  if curr.next.CAS (unmarked succ, marked succ) then
  |  |  |  pred.next.CAS (unmarked curr, unmarked succ)
  |  |  |  return true
  |  |  end
  |  end
  end
```

```
Procedure Find(int key)
  Data: Node* curr = head
  while curr.key < key do
  |  curr = curr.next
  end
  return (curr.key == key && curr.next is unmarked)
```

Figure 1: Harris Non-Blocking Algorithm

1.0.1 Crash-Recovery

The linearization point 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 of the procedure return, that is, either when curr.key != key, or at the second condition test.

Following these linearization points (committing proof...), insert and delete operation are linearized at the point where they affect the system. That is, if there is a linearization point for insert operation, then all process will see the new node starting from this point, and if a node was logically removed, then all processes treat it as a removed node. 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 and affect all other processes, or it did not affect the data structure at all.

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 at line 37 after performing a successful CAS. 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, as it is no longer part of the list.

1.0.2 Linked-List Recoverable Version

To solve the problems mention above, we present a modification for the algorithm such that in case a process fails, upon recovery it is able to complete its last pending operation and also return the response value.

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 a process p successfully mark a node (logical delete), it tries to write its id to deleter using CAS. This way, if a process fails during a delete, it can use deleter in order to determine the response value. We assume deleter is initialized 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.

We present the modified algorithm. Only the procedures which require changes are presented. For simplicity, a process creates a new operation structure each time it writes to Backup, although a process can use two such structures alternately.

Correctness Argument

In the following, we give an intuition for the correctness of the algorithm.

For the Insert operation, p tries to add the new node by performing a CAS. If it succeeds it will return true if it suffers no failure. In case of a failure after writing to Backup[p], upon recovery p tries to complete its operation. If it already performed a successful CAS, that is, the node was added to the list, then either it is still in the list or that it was deleted. Therefore, if p can find the new node in the list (using a procedure similar to find), or that it is marked, it must be that the node was added, and p can return true. Otherwise, p either crashed before performing the CAS, or that the CAS was unsuccessful. In both cases, the new node was not added, and p can restart the Insert procedure.

For the delete operation, once p logically delete a node v, it also tries to announce itself as the "removal" of the node by writing its id to deleter using CAS. Assume a process p crash while trying to delete the node. Upon recovery, if p sees the node is not marked, then obliviously its deletion did not took affect, and it can restart the delete operation. However, if the node is marked, it might be that p marked it before the crash, or it might be some other process trying to delete the same node did so. As p can not distinguish between the two, and since we desire for a lock-free implementation, we let p to try and complete the deletion, even if it is not to process to logically delete the node. To avoid a scenario in which more then a single process "delete" the same node, they all compete for deleter using CAS. The first one to perform it will win, and it is the only process to return true. It is easy to verify once a process writes to deleter, then eventually, if given enough time with no crash, it returns true, while any other process trying to delete the same node will have to retry the delete operation.

Procedure Recover

```
if Backup[p].type == Insert then
    if Backup[p].new is in the list || Backup[p].curr.next is marked then
        return true
    else
        go to 47                                        // restart Insert
    end
end
if Backup[p].type == Delete then
    if Backup[p].curr.next is marked then
        go to 67                     // try to complete the deletation
    else
        go to 59                                        // restart Delete
    end
end
```

2 Elimination Stack

Procedure Exchange(⟨T,flag⟩ slot, T myitem, long timeout)

```
Data: long timeBound = getNanos() + timeout
while true do
    if getNanos() > timeBound then
        return TIMEOUT
    end
    ⟨youritem,state⟩ = slot
    switch state do
        case EMPTY do
            if slot.CAS (⟨youritem,EMPTY⟩, ⟨myitem, WAITING⟩) then
                while getNanos() < timeBound do
                    ⟨youritem,state⟩ = slot
                    if state == BUSY then
                        slot = ⟨null, EMPTY⟩
                        return youritem
                    end
                end
                if slot.CAS (⟨myitem,WAITING⟩, ⟨null,EMPTY⟩) then
                    return TIMEOUT
                else
                    ⟨youritem,state⟩ = slot
                    slot = ⟨null, EMPTY⟩
                    return youritem
                end
            end
            break
        end
        case WAITING do
            if slot.CAS (⟨youritem,state⟩, ⟨myitem,BUSY⟩) then
                return youritem
            end
            break
        end
        case BUSY do
            break
        end
    end
end
```

Procedure TryPush(Node newNode)

```
Node* oldTop = Top
newNode.next = oldTop
if Top.CAS (oldTop, newNode) then
    return true
else
    return false
end
```

Procedure TryPop(void)

```
Node *oldTop = Top
Node *newTop
if oldTop == NULL then
    return EMPTY
end
newTop = oldTop.next
if Top.CAS (oldTop, newTop) then
    return oldTop
end
return NULL
```

3 BST

The original BST algorithm does not support the crash-recovery model. It is clear from the code a process does not persist an operation response in the non-volatile memory, and thus, once a process crash the response is lost. For example, assume a process p apply INSERT(k) and assume p performs

Procedure push(T myitem)

```
Node *nd = new Node (myitem)
while true do
    if TryPush(nd) then
        return true
    end
    othervalue = visit(nd)
    if othervalue == NULL then
        Record Success ()
        return true
    else
        Record Failure ()
    end
end
```

a successful CAS in line 3 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 a Find(k) operation will return it. However, even though the operation was already linearized at the time of the crash, upon recovery process p 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 is supplied, it may result an execution which is not well-formed. Consider for example the following scenario. A process p invoke an Op_1 = Insert(k_1) operation. After a successful CAS at line 3 the process crush. After recovering, p invoke an Op_2 = Insert(k_2) operation. Assume k_1 and k_2 belongs to a completely different parts of the tree. Then, p 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 process accessing 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. In the Linked-List algorithm it was enough to consider a marked node as if it has been already deleted. Nonetheless, the more complex BST implementation is even 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 complete the delete operation, and only then will return, according to whether the attempt was successful. The modified Find routine is given in figure 6.

solved by having the Find routine helping in case of a flagged or marked node. In such case, once a process flagged a node, then any operation, including Find, will try and complete this operation, and thus either the operation is linearized at the time of the flagging, or there is no linearization at all. A more fine modification will be as follows. The routine Find calls Search and gets $\langle gp, p, l, pupdate, gpupdate \rangle$ in return. Then the process look to see if there is an unavoidable operation in progress that may affects its response. There are several cases to consider:
$l \rightarrow key \neq k$, then return Falseif and only if *pupdate* does not contains indication for inserting a key k.
$l \rightarrow key = k$, then if nor *gpupdate* neither *pupdate* contains indication for deleting l it is safe to return True; Else, if $pupdate.state$ = Mark and $pupdate.info$'s Leaf is l, then return False; Else, $gpupdate.state$ = DFlag, and $gpupdate.info$'s Leaf is l. In such case, we do not know if the deletion will be completed or not (resulting a response Falseor True, accordingly), so first need to try and complete it. Hence, a HelpDelete routine with *gpupdate* is invoked. Once it completes, the response is whether the deletion took affect or not, that is, let $par := gpupdate.info \leftarrow p$, then it returns the evaluation of the statement $par \leftarrow update.state$ = Mark.

The above solution prevents unwanted behaviour so that

However, such a solution is not efficient for the Find routine, as it needs to complete other operations.

The helping mechanism, in which a process performing Insert or Delete first helps operation which marked of flagged the relevant nodes, guarantees the BST implementation is consistent even in the presence of crush-recovery. Any operation has a single linearization point, which is the point where the successful CAS changes the tree data structure, that is, the CAS at the CAS-Child routine. Therefore, even in the case of a crush,

If a process p crush in a middle of executing an operation Op, then either Op did not flagged or marked any node, thus it did not took affect on the data-structure, nor it will in the future. However, if Op already flagged or marked a node, then the helping mechanism guarantees that

Shared variables: Node* head

Define Info: struct { *type*: OperationType, *pred*,*curr*, *new*: Node* }

Code for process p:

```
Procedure Insert(int key)
  Data: Node* pred, curr
        Node node = new Node (key)
  while true do
      <pred, curr> = lookup(key)
      if curr.key == key then
          return false
      else
          node.next = unmarked curr
          Backup[p] = new Info (Insert, pred, curr, new)
          if pred.next.CAS (unmarked curr, unmarked node) then
              return true
          end
      end
  end
```

```
Procedure Delete(int key)
  Data: Node* pred, curr, succ
  while true do
      <pred, curr> = lookup(key)
      if curr.key != key then
          return false
      else
          succ = curr.next
          Backup[p] = new Info (Delete, pred, curr, null)
          if curr.next.CAS (unmarked succ, marked succ) then
              curr.deleter.CAS (null, p)
              pred.next.CAS (unmarked curr, unmarked succ)
              return (curr.deleter == p)
          end
      end
  end
```

Figure 2: Recoverable Non-Blocking Linked-List

type Update { ▷ stored in one CAS word
    $\{\textsc{Clean}, \text{DFlag}, \text{IFlag}, \textsc{Mark}\}$ *state*
    Flag **info*
}
type Internal { ▷ subtype of Node
    $Key \cup \{\infty_1, \infty_2\}$ *key*
    Update *update*
    Node **left*, **right*
}
type Leaf { ▷ subtype of Node
    $Key \cup \{\infty_1, \infty_2\}$ *key*
}
type IInfo { ▷ subtype of Flag
    Internal **p*, **newInternal*
    Leaf **l*
}
type DInfo { ▷ subtype of Flag
    Internal **gp*, **p*
    Leaf **l*
    Update *pupdate*
}
▷ Initialization:
shared Internal **Root* := pointer to new Internal node with *key* field ∞_2, *update* field $\langle \textsc{Clean}, \perp \rangle$, and pointers to new Leaf nodes with keys ∞_1 and ∞_2, respectively, as *left* and *right* fields.

Figure 3: Type definitions and initialization.

```
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⟩
            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
    CAS(op → p → update, ⟨IFlag, op⟩, ⟨Clean, op⟩)            ▷ iunflag CAS
}
```

Figure 4: 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⟩
            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
    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 5: Pseudocode for Delete and some auxiliary routines.

```
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 pupdate.info ← key k
        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 6: 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 7: Type definitions and initialization.

```
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[p] := 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 8: 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[p] := 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 9: Pseudocode for Delete and some auxiliary routines.