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

1 retry: **while true do**
2   pred = head
3   curr = head.next
4   **while true do**
5   succ = curr.next
6   **if** *curr.next is marked* **then**
7   **if** *pred.next.***CAS** *(unmarked curr,unmarked succ)* == **false then**
8   **go to** retry
9   **end**
10   curr = succ
11   **else**
12   **if** *curr.key* $\geq$ *key* **then**
13   **return** $\langle$pred,curr$\rangle$
14   **end**
15   pred = curr
16   curr = succ
17   **end**
18   **end**
19 **end**

Shared variables: Node* head

Procedure Insert(int key)

    Data: Node* pred, curr
           Node node = **new** Node (key)

20 while true do
21   ⟨pred, curr⟩ = lookup(key)
22   **if** *curr.key == key* **then**
23   **return false**
24   **else**
25   node.next = unmarked curr
26   **if** *pred.next.***CAS** *(unmarked curr, unmarked node)* **then**
27   **return true**
28   **end**
29   **end**
30 end

Procedure Delete(int key)

    Data: Node* pred, curr, succ

31 while true do
32   ⟨pred, curr⟩ = lookup(key)
33   **if** *curr.key != key* **then**
34   **return false**
35   **else**
36   succ = curr.next
37   **if** *curr.next.***CAS** *(unmarked succ, marked succ)* **then**
38   pred.next.**CAS** (unmarked curr, unmarked succ)
39   **return true**
40   **end**
41   **end**
42 end

Procedure Find(int key)

    Data: Node* curr = head
43 while *curr.key < key* **do**
44   curr = curr.next
45 end
46 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

73	**if** $Backup[p].type = Insert$ **then**
74	**if** $Backup[p].new$ is in the list $||$ $Backup[p].curr.next$ is marked **then**
75	**return true**
76	**else**
77	**go to** 47 // restart Insert
78	
79	**end**
80	**end**
81	**if** $Backup[p].type == Delete$ **then**
82	**if** $Backup[p].curr.next$ is marked **then**
83	**go to** 67 // try to complete the deletation
84	
85	**else**
86	**go to** 59 // restart Delete
87	
88	**end**
89	**end**

2 Elimination Stack

Procedure Exchange($\langle$T,flag$\rangle$ slot, T myitem, long timeout)

Data: long timeBound = getNanos() + timeout

```
90  while true do
91  │   if getNanos() > timeBound then
92  │   │   return TIMEOUT
93  │   end
94  │   ⟨youritem,state⟩ = slot
95  │   switch state do
96  │   │   case EMPTY
97  │   │   │   if slot.CAS (⟨youritem,EMPTY⟩, ⟨myitem, WAITING⟩) then
98  │   │   │   │   while getNanos() < timeBound do
99  │   │   │   │   │   ⟨youritem,state⟩ = slot
100 │   │   │   │   │   if state == BUSY then
101 │   │   │   │   │   │   slot = ⟨null, EMPTY⟩
102 │   │   │   │   │   │   return youritem
103 │   │   │   │   │   end
104 │   │   │   │   end
105 │   │   │   │   if slot.CAS (⟨myitem,WAITING⟩, ⟨null,EMPTY⟩) then
106 │   │   │   │   │   return TIMEOUT
107 │   │   │   │   else
108 │   │   │   │   │   ⟨youritem,state⟩ = slot
109 │   │   │   │   │   slot = ⟨null, EMPTY⟩
110 │   │   │   │   │   return youritem
111 │   │   │   │   end
112 │   │   │   end
113 │   │   │   break
114 │   │   end
115 │   │   case WAITING
116 │   │   │   if slot.CAS (⟨youritem,state⟩, ⟨myitem,BUSY⟩) then
117 │   │   │   │   return youritem
118 │   │   │   end
119 │   │   │   break
120 │   │   end
121 │   │   case BUSY
122 │   │   │   break
123 │   │   end
124 │   endsw
125 end
```

Procedure TryPush(Node newNode)

---

126 Node* oldTop = Top
127 newNode.next = oldTop
128 **if** *Top.***CAS** *(oldTop, newNode)* **then**
129 | **return true**
130 **else**
131 | **return false**
132 **end**

Procedure TryPop(void)

---

133 Node *oldTop = Top
134 Node *newTop
135 **if** *oldTop == NULL* **then**
136 | **return** EMPTY
137 **end**
138 newTop = oldTop.next
139 **if** *Top.***CAS** *(oldTop, newTop)* **then**
140 | **return** oldTop
141 **end**
142 **return** NULL

---

3 BST

Procedure push(T myitem)

---

143 Node *nd = new Node (myitem)
144 **while true do**
145 | **if** *TryPush(nd)* **then**
146 | | **return true**
147 | **end**
148 | othervalue = visit(nd)
149 | **if** *othervalue == NULL* **then**
150 | | Record Success ()
151 | | **return true**
152 | **else**
153 | | Record Failure ()
154 | **end**
155 **end**

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)

47 **while** true **do**
48   <pred, curr> = lookup(key)
49   **if** *curr.key == key* **then**
50   **return false**
51   **else**
52   node.next = unmarked curr
53   Backup[p] = new Info (Insert, pred, curr, new)
54   **if** *pred.next.***CAS** *(unmarked curr, unmarked node)* **then**
55   **return true**
56   **end**
57   **end**
58 **end**

Procedure Delete(int key)

    Data: Node* pred, curr, succ

59 **while** true **do**
60   <pred, curr> = lookup(key)
61   **if** *curr.key != key* **then**
62   **return false**
63   **else**
64   succ = curr.next
65   Announe[p] = new Info (Delete, pred, curr, null)
66   **if** *curr.next.***CAS** *(unmarked succ, marked succ)* **then**
67   curr.deleter.**CAS** (null, p)
68   pred.next.**CAS** (unmarked curr, unmarked succ)
69   **return** (curr.deleter == p)
70   **end**
71   **end**
72 **end**

Figure 2: Recoverable Non-Blocking Linked-List

```
1    type Update {                    ▷ stored in one CAS word
2        {CLEAN, DELETEFLAG, INSERTFLAG, MARK} state
3        Flag *info
4    }
5    type Internal {                  ▷ subtype of Node
6        Key ∪ {∞₁, ∞₂} key
7        Update update
8        Node *left, *right
9    }
10   type Leaf {                      ▷ subtype of Node
11       Key ∪ {∞₁, ∞₂} key
12   }
13   type IFlag {                     ▷ subtype of Flag
14       Internal *p, *newInternal
15       Leaf *l
16   }
17   type DFlag {                     ▷ subtype of Flag
18       Internal *gp, *p
19       Leaf *l
20       Update pupdate
21   }
     ▷ Initialization:
22   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.

23 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 $p \to left$ has contained l (if $k < p \to key$) or $p \to right$ has contained l (if $k \geq p \to key$)
        ▷ (3) $p \to update$ has contained *pupdate*
        ▷ (4) if $l \to key \neq \infty_1$, then the following three statements hold:
        ▷   (4a) gp points to an Internal node
        ▷   (4b) either $gp \to left$ has contained p (if $k < gp \to key$) or $gp \to right$ has contained p (if $k \geq gp \to key$)
        ▷   (4c) $gp \to update$ has contained *gpupdate*
24   Internal *gp, *p
25   Node *$l := Root$
26   Update *gpupdate, pupdate*   ▷ Each stores a copy of an *update* field

27   while l points to an internal node {
28          $gp := p$ ▷ Remember parent of p
29          $p := l$ ▷ Remember parent of l
30          $gpupdate := pupdate$ ▷ Remember *update* field of gp
31          $pupdate := p \to update$ ▷ Remember *update* field of p
32   if $k < l \to key$ then $l := p \to left$ else $l := p \to right$ ▷ Move down to appropriate child
33   }
34   return ⟨$gp, p, l, pupdate, gpupdate$⟩
35 }

36 FIND(*Key k*) : Leaf* {
37   Leaf *l

38   ⟨−, −, l, −, −⟩ := SEARCH(k)
39   if $l \to key = k$ then return l
40   else return ⊥
41 }

42 INSERT(*Key k*) : boolean {
43   Internal *p, *$newInternal$
44   Leaf *l, *$newSibling$
45   Leaf *new := pointer to a new Leaf node whose *key* field is k
46   Update *pupdate, result*
47   IFlag *op

48   while TRUE {
49   ⟨−, p, l, *pupdate*, −⟩ := SEARCH(k)
50   if $l \to key = k$ then return FALSE   ▷ Cannot insert duplicate key
51   if *pupdate.state* ≠ CLEAN then HELP(*pupdate*)   ▷ Help the other operation
52   else {
53              $newSibling$:= pointer to a new Leaf whose key is $l \to key$
54              $newInternal$:= pointer to a new Internal node with *key* field max($k, l \to key$),
                    update field ⟨CLEAN, ⊥⟩, and with two child fields equal to *new* and *newSibling*
                    (the one with the smaller key is the left child)
55   op := pointer to a new IFlag record containing ⟨$p, l, newInternal$⟩
56              $result$:= CAS($p \to update$, *pupdate*, ⟨INSERTFLAG, op⟩)   ▷ **iflag** CAS
57   if $result = pupdate$ then {   ▷ The iflag CAS was successful
58   HELPINSERT(op)   ▷ Finish the insertion
59   return TRUE
60   }
61   else HELP($result$)   ▷ The iflag CAS failed; help the operation that caused failure
62   }
63   }
64 }

65 HELPINSERT(IFlag *op) {
        ▷ *Precondition*: *op* points to an IFlag record (*i.e.*, it is not ⊥)
66   CAS-CHILD($op \to p$, $op \to l$, $op \to newInternal$)   ▷ **ichild** CAS
67   CAS($op \to p \to update$, ⟨INSERTFLAG, op⟩, ⟨CLEAN, op⟩)   ▷ **iunflag** CAS
68 }

Figure 4: Pseudocode for SEARCH, FIND and INSERT.

9

```
69  DELETE(Key k) : boolean {
70      Internal *gp, *p
71      Leaf *l
72      Update pupdate, gpupdate, result
73      DFlag *op

74      while TRUE {
75          ⟨gp, p, l, pupdate, gpupdate⟩ := SEARCH(k)
76          if l → key ≠ k then return FALSE                                    ▷ Key k is not in the tree
77          if gpupdate.state ≠ CLEAN then HELP(gpupdate)
78          else if pupdate.state ≠ CLEAN then HELP(pupdate)
79          else {                                                             ▷ Try to flag gp
80              op := pointer to a new DFlag record containing ⟨gp, p, l, pupdate⟩
81              result := CAS(gp → update, gpupdate, ⟨DELETEFLAG, op⟩)          ▷ dflag CAS
82              if result = gpupdate then {                                     ▷ CAS successful
83                  if HELPDELETE(op) then return TRUE                          ▷ Either finish deletion or unflag
84              }
85              else HELP(result)                  ▷ The dflag CAS failed; help the operation that caused the failure
86          }
87      }
88  }

89  HELPDELETE(DFlag *op) : boolean {
        ▷ Precondition: op points to a DFlag record (i.e., it is not ⊥)
90      Update result                                                          ▷ Stores result of mark CAS

91      result := CAS(op → p → update, op → pupdate, ⟨MARK, op⟩)               ▷ mark CAS
92      if result = op → pupdate or result = ⟨MARK, op⟩ then {                  ▷ op → p is successfully marked
93          HELPMARKED(op)                                                     ▷ Complete the deletion
94          return TRUE                                                        ▷ Tell DELETE routine it is done
95      }
96      else {                                                                 ▷ The mark CAS failed
97          HELP(result)                                                       ▷ Help operation that caused failure
98          CAS(op → gp → update, ⟨DELETEFLAG, op⟩, ⟨CLEAN, op⟩)               ▷ backtrack CAS
99          return FALSE                                                       ▷ Tell DELETE routine to try again
100     }
101 }

102 HELPMARKED(DFlag *op) {
        ▷ Precondition: op points to a DFlag record (i.e., it is not ⊥)
103     Node *other

        ▷ Set other to point to the sibling of the node to which op → l points
104     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
105     CAS-CHILD(op → gp, op → p, other)                                      ▷ dchild CAS
106     CAS(op → gp → update, ⟨DELETEFLAG, op⟩, ⟨CLEAN, op⟩)                   ▷ dunflag CAS
107 }

108 HELP(Update u) {                                                           ▷ General-purpose helping routine
        ▷ Precondition: u has been stored in the update field of some internal node
109     if u.state = INSERTFLAG then HELPINSERT(u.info)
110     else if u.state = MARK then HELPMARKED(u.info)
111     else if u.state = DELETEFLAG then HELPDELETE(u.info)
112 }

113 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.
114     if new → key < parent → key then
115         CAS(parent → left, old, new)
116     else
117         CAS(parent → right, old, new)
118 }
```

Figure 5: Pseudocode for DELETE and some auxiliary routines.

```
1    type Update {              ▷ stored in one CAS word
2         {CLEAN, DELETEFLAG, INSERTFLAG, MARK} state
3         Flag *info
4    }
5    type Internal {            ▷ subtype of Node
6         Key ∪ {∞₁, ∞₂} key
7         Update update
8         Node *left, *right
9    }
10   type Leaf {                ▷ subtype of Node
11        Key ∪ {∞₁, ∞₂} key
12   }
13   type IFlag {               ▷ subtype of Flag
14        Internal *p, *newInternal
15        Leaf *l
16        boolean complete
17   }
18   type DFlag {               ▷ subtype of Flag
19        Internal *gp, *p
20        Leaf *l
21        Update pupdate
22        boolean complete
23   }
     ▷ Initialization:
24   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 6: Type definitions and initialization.

25   SEARCH($Key\ k$) : $\langle$Internal*, Internal*, Leaf*, Update, Update$\rangle$ {
       ▷ 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 $p \to left$ has contained l (if $k < p \to key$) or $p \to right$ has contained l (if $k \geq p \to key$)
       ▷ (3) $p \to update$ has contained *pupdate*
       ▷ (4) if $l \to key \neq \infty_1$, then the following three statements hold:
       ▷   (4a) gp points to an Internal node
       ▷   (4b) either $gp \to left$ has contained p (if $k < gp \to key$) or $gp \to right$ has contained p (if $k \geq gp \to key$)
       ▷   (4c) $gp \to update$ has contained *gpupdate*
26   Internal *gp, *p
27   Node *$l := Root$
28   Update $gpupdate, pupdate$ ▷ Each stores a copy of an *update* field

29   while l points to an internal node {
30          $gp := p$ ▷ Remember parent of p
31          $p := l$ ▷ Remember parent of l
32          $gpupdate := pupdate$ ▷ Remember *update* field of gp
33          $pupdate := p \to update$ ▷ Remember *update* field of p
34   if $k < l \to key$ then $l := p \to left$ else $l := p \to right$ ▷ Move down to appropriate child
35   }
36   return $\langle gp, p, l, pupdate, gpupdate \rangle$
37 }

38 FIND($Key\ k$) : Leaf* {
39   Leaf *l

40       $\langle -, -, l, -, - \rangle := $ SEARCH(k)
41   if $l \to key = k$ then return l
42   else return $\perp$
43 }

44 INSERT($Key\ k$) : boolean {
45   Internal *p, *$newInternal$
46   Leaf *l, *$newSibling$
47   Leaf *$new := $ pointer to a new Leaf node whose key field is k
48   Update $pupdate, result$
49   IFlag *op

50   while TRUE {
51          $\langle -, p, l, pupdate, - \rangle := $ SEARCH(k)
52   if $l \to key = k$ then return FALSE   ▷ Cannot insert duplicate key
53   if $pupdate.state \neq$ CLEAN then HELP($pupdate$)   ▷ Help the other operation
54   else {
55             $newSibling := $ pointer to a new Leaf whose key is $l \to key$
56             $newInternal := $ pointer to a new Internal node with key field $\max(k, l \to key)$,
                $update$ field $\langle$CLEAN, $\perp\rangle$, and with two child fields equal to new and $newSibling$
                (the one with the smaller key is the left child)
57             $op := $ pointer to a new IFlag record containing $\langle p, l, newInternal \rangle$
58             $result := $ CAS($p \to update, pupdate, \langle$INSERTFLAG, $op\rangle$)   ▷ **iflag** CAS
59   if $result = pupdate$ then {   ▷ The iflag CAS was successful
60   HELPINSERT(op)   ▷ Finish the insertion
61   return TRUE
62   }
63   else HELP($result$)   ▷ The iflag CAS failed; help the operation that caused failure
64   }
65   }
66 }

67 HELPINSERT(IFlag *op) {
       ▷ *Precondition*: op points to an IFlag record (*i.e.*, it is not $\perp$)
68   CAS-CHILD($op \to p, op \to l, op \to newInternal$)   ▷ **ichild** CAS
69   CAS($op \to p \to update, \langle$INSERTFLAG, $op\rangle, \langle$CLEAN, $op\rangle$)   ▷ **iunflag** CAS
70 }

Figure 7: Pseudocode for SEARCH, FIND and INSERT.

```
71  DELETE(Key k) : boolean {
72       Internal *gp, *p
73       Leaf *l
74       Update pupdate, gpupdate, result
75       DFlag *op

76       while TRUE {
77            ⟨gp, p, l, pupdate, gpupdate⟩ := SEARCH(k)
78            if l → key ≠ k then return FALSE                              ▷ Key k is not in the tree
79            if gpupdate.state ≠ CLEAN then HELP(gpupdate)
80            else if pupdate.state ≠ CLEAN then HELP(pupdate)
81            else {                                                         ▷ Try to flag gp
82                 op := pointer to a new DFlag record containing ⟨gp, p, l, pupdate⟩
83                 result := CAS(gp → update, gpupdate, ⟨DELETEFLAG, op⟩)    ▷ dflag CAS
84                 if result = gpupdate then {                              ▷ CAS successful
85                      if HELPDELETE(op) then return TRUE                   ▷ Either finish deletion or unflag
86                 }
87                 else HELP(result)                 ▷ The dflag CAS failed; help the operation that caused the failure
88            }
89       }
90  }

91  HELPDELETE(DFlag *op) : boolean {
         ▷ Precondition: op points to a DFlag record (i.e., it is not ⊥)
92       Update result                                                     ▷ Stores result of mark CAS

93       result := CAS(op → p → update, op → pupdate, ⟨MARK, op⟩)           ▷ mark CAS
94       if result = op → pupdate or result = ⟨MARK, op⟩ then {            ▷ op → p is successfully marked
95            HELPMARKED(op)                                                ▷ Complete the deletion
96            return TRUE                                                   ▷ Tell DELETE routine it is done
97       }
98       else {                                                            ▷ The mark CAS failed
99            HELP(result)                                                  ▷ Help operation that caused failure
100           CAS(op → gp → update, ⟨DELETEFLAG, op⟩, ⟨CLEAN, op⟩)          ▷ backtrack CAS
101           return FALSE                                                  ▷ Tell DELETE routine to try again
102      }
103 }

104 HELPMARKED(DFlag *op) {
         ▷ Precondition: op points to a DFlag record (i.e., it is not ⊥)
105      Node *other

         ▷ Set other to point to the sibling of the node to which op → l points
106      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
107      CAS-CHILD(op → gp, op → p, other)                                  ▷ dchild CAS
108      CAS(op → gp → update, ⟨DELETEFLAG, op⟩, ⟨CLEAN, op⟩)               ▷ dunflag CAS
109 }

110 HELP(Update u) {                                                       ▷ General-purpose helping routine
         ▷ Precondition: u has been stored in the update field of some internal node
111      if u.state = INSERTFLAG then HELPINSERT(u.info)
112      else if u.state = MARK then HELPMARKED(u.info)
113      else if u.state = DELETEFLAG then HELPDELETE(u.info)
114 }

115 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.
116      if new → key < parent → key then
117           CAS(parent → left, old, new)
118      else
119           CAS(parent → right, old, new)
120 }
```

Figure 8: Pseudocode for DELETE and some auxiliary routines.