```
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 1: 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 → 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
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 → update                                  ▷ Remember update field of p
32         if k < l → key then l := p → left else l := p → 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 → 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 → 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 → key
54             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)
55             op := pointer to a new IFlag record containing ⟨p, l, newInternal⟩
56             result := CAS(p → 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 → p, op → l, op → newInternal)                ▷ ichild CAS
67     CAS(op → p → update, ⟨INSERTFLAG, op⟩, ⟨CLEAN, op⟩)        ▷ iunflag CAS
68 }
```


Figure 2: Pseudocode for SEARCH, FIND and INSERT

```
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 }
```