
Algorithm 1: structures used

```

1 struct Node{
2   {Boolean,key} markAndKey;
3   {Boolean,Boolean,Boolean,NodePtr} child[2];
4   Boolean readyToReplace;
5 };
6 struct seekRecord{
7   NodePtr node; NodePtr parent;
8   NodePtr lastUParent; NodePtr lastUNode;
9   NodePtr injectionPoint;
10 };
11 struct State{
12   NodePtr node; NodePtr parent;
13   Key key;
14   enum mode{ INJECTION, DISCOVERY, CLEANUP };
15   enum type{ SIMPLE, COMPLEX };
16   seekRecPtr seekRec;
17 };

```

Algorithm 2: Search(*key*)

```

18 seek( key, mySeekRec);
19  $\langle *, nKey \rangle := mySeekRec \rightarrow node \rightarrow markAndKey$ ;
20 if key = nKey then return true;
21 else return false;

```

Algorithm 3: Insert(*key*)

```

22 while true do
23   seek( key, mySeekRec);
24    $\langle *, nKey \rangle := mySeekRec \rightarrow node \rightarrow markAndKey$ ;
25   if key = nKey then return false;
26   newNode:= create a new node and initialize its fields;
27   which := key < nKey ? LEFT: RIGHT;
28    $\langle *, *, *, address \rangle := mySeekRec \rightarrow injectionPoint$ ;
29   out:= CAS( $node \rightarrow child[which]$ ,  $\langle 1, 0, 0, address \rangle$ ,  $\langle 0, 0, 0, newNode \rangle$ );
30   if out then return true;
31    $\langle *, d, p, address \rangle := node \rightarrow child[which]$ ; // find out why the CAS failed
32   if not (d or p) then continue; // CAS failed due to another insert op
33   deepHelp( $mySeekRec \rightarrow lastUNode$ ,  $mySeekRec \rightarrow lastUParent$ );

```

Algorithm 4: Delete(*key*)

```

// initialize the state record
34 myState→mode:= INJECTION; myState→key:= key;
35 while true do
36   seek( key, mySeekRec);
37   node:= mySeekRec→node; parent:= mySeekRec→parent;
38   ⟨*, nKey⟩ := node→markAndKey;
39   if myState→key≠ nKey then
40     // the key does not exist in the tree
41     if myState→mode= INJECTION then return false;
42     else return true;
43   needToHelp:=false;
44   // perform appropriate action depending on the mode
45   if myState→mode= INJECTION then
46     myState→node:= node // store a reference to the node
47     out:= inject(myState) // attempt to inject
48     if not out then needToHelp:= true;
49   // mode would have changed if the op was injected
50   if myState→mode≠ INJECTION then
51     // if the node found by seek is different from the one stored
52     // in state record, then the node is already deleted
53     if myState→node≠ node then return true;
54     myState→parent:= parent // update parent with recent seek
55   if myState→mode= DISCOVERY then
56     findAndMarkSuccessor(myState);
57   if myState→mode= DISCOVERY then
58     removeSuccessor(myState);
59   if myState→mode= CLEANUP then
60     out:= cleanup(myState,0);
61     if out then return true;
62     else
63       ⟨*, nKey⟩ := node→markAndKey; myState→key:= nKey;
64       // help if helpee node is not the node of interest
65       if mySeekRec→lastUNode≠ node then needToHelp:=true;
66   if needToHelp then
67     deepHelp(mySeekRec→lastUNode, mySeekRec→lastUParent) ;

```

Algorithm 5: Inject(*state*)

```

61 node := state → node // try to set the delete flag on the left edge
62 while true do
63    $\langle n, d, p, left \rangle := node \rightarrow child[LEFT]$ ;
64   if d or p then return false; // edge is already marked
65   out := CAS(node → child[LEFT],  $\langle n, 0, 0, left \rangle$ ,  $\langle n, 1, 0, left \rangle$ );
66   if out then break; // retry from beginning of while loop
67   updateModeAndType(state) // mark right edge, update mode and type
68   return true;

```

Algorithm 6: updateModeAndType(*state*)

```

69 node := state → node // retrieve the address from the state record
70 if node → child[RIGHT] ≠  $\langle *, 1, *, * \rangle$  then // mark right edge if unmarked
71   BTS(node → child[RIGHT], DELETE_FLAG);
72  $\langle m, * \rangle := node \rightarrow markAndKey$ ;
73  $\langle lN, *, *, * \rangle := node \rightarrow child[LEFT]$ ;  $\langle rN, *, *, * \rangle := node \rightarrow child[RIGHT]$ ;
74 if lN or rN then // update the op mode and type
75   if m then
76      $state \rightarrow type := COMPLEX$ ; node → readyToReplace := true;
77   else
78      $state \rightarrow type := SIMPLE$ ; state → mode := CLEANUP;
79 else
80    $state \rightarrow type := COMPLEX$ ;
81   if readyToReplace then state → mode := CLEANUP;
82   else state → mode := DISCOVERY;
83 return ;

```

Algorithm 7: findSmallest(*node*, *right*, *seekRec*)

```

// find the smallest key in the subtree rooted at the right child
84 lastUParent := node; lastUNode := right; prev := node; curr := right;
85 while true do
86    $\langle n, d, p, left \rangle := curr \rightarrow child[LEFT]$ ;
87   if n then break;
88   prev := curr; curr := left; // traverse the next edge
89   if not (d or p) then // keep track of the last unmarked edge
90      $lastUParent := prev$ ; lastUNode := curr
91   // update the seek record
91   return;

```

Algorithm 8: cleanup(*state*, *dFlg*)

```

// retrieve the addresses from the state record
92 pWhich := edge of the parent which needs to be switched;
93 if state → type = COMPLEX then
94   newNode := a new copy of the node in which all the fields are unmarked;
   // try to switch the edge at the parent
95   out := CAS(parent → child[pWhich], ⟨0, dFlg, 0, node⟩, ⟨0, dFlg, 0, newNode⟩);
96 else
97   nWhich := non-Null child of the node being deleted;
98   ⟨n, *, *, address⟩ := node → child[nWhich];
99   if n then // set only the null flag; do not change the address
100     out := CAS(parent → child[pWhich], ⟨0, dFlg, 0, node⟩, ⟨1, dFlg, 0, node⟩);
101   else // change the address here by switching the pointer
102     out := CAS(parent → child[pWhich], ⟨0, dFlg, 0, node⟩, ⟨0, dFlg, 0, address⟩);
103 return out;

```

Algorithm 9: seek(*key*, *seekRec*)

```

104 while true do
   // create two local seek records: cSeek (current seek record) and
   // pSeek (previous seek record) used for the traversal
105   while true do
106     ⟨*, cKey⟩ := curr → markAndKey; // key in the curr of cSeek
107     if key = cKey then // key found; stop the traversal
108       done := true; break;
109     which := key < cKey ? LEFT : RIGHT;
110     ⟨n, d, p, address⟩ := curr → child[which]; // read the next edge
111     if n then // null flag is set; reached a leaf node
112       if key stored in anchorNode has not changed then
113         done := true; break; // use data from cSeek
114       else if anchorNode of cSeek & pSeek matches then
115         done := true; break; // use data from pSeek
116       else
117         break; // after copying cSeek to pSeek
118     if which = RIGHT then // next edge to be traversed is a right edge
119       anchorNode := curr; // keep track of curr node
120       anchorKey := cKey; // and its key
121     prev := curr; curr := address; // traverse the next edge
122     if not (d or p) then // keep track the last unmarked edge
123       lastUParent := prev; lastUNode := curr;
124   if done then
   // initialize the appropriate seek record (cSeek or pSeek)
125   return;

```
