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;
8      NodePtr parent;
9      NodePtr lastUParent;
10     NodePtr lastUNode;
11 };
12 struct State{
13     NodePtr node;
14     NodePtr parent;
15     Key key;
16     enum mode{ INJECTION, DISCOVERY, CLEANUP };
17     enum type{ SIMPLE, COMPLEX };
18     seekRecPtr seekRec;
19 };
```

Algorithm 2: Search(key)

```
20 seek( key, mySeekRec);
21 ⟨*, nKey⟩ := mySeekRec→node→markAndKey;
22 if key = nKey then return true;
23 else return false;
```

Algorithm 3: Insert(key)

```
24 while true do
25     seek( key, mySeekRec);
26     ⟨*, nKey⟩ := mySeekRec→node→markAndKey;
27     if key = nKey then return false;
28     newNode:= create a new node and initialize its fields;
29     which := key < nKey ? LEFT: RIGHT;
30     ⟨*,*,*,address⟩ := node→child[which];
31     result:= CAS(node→child[which],⟨1,0,0,address⟩,⟨0,0,0,newNode⟩);
32     if result then return true;
33     ⟨*,d,p,address⟩ := node→child[which];
34     if not (d or p) then continue;
35     deepHelp(mySeekRec→lastUNode,mySeekRec→lastUParent);
```

Algorithm 4: Delete(key)

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

Algorithm 5: Inject(*state*)

```
63 node:= state→node // try to set the delete flag on the left edge
64 while true do
65 |  ⟨n,d,p,left⟩ := node→child[LEFT];
66 |  if d or p then return false; // edge is already marked
67 |  result:= CAS(node→child[LEFT],⟨n,0,0,left⟩,⟨n,1,0,left⟩);
68 |  if result then break; // retry from beginning of while loop
69 |  updateModeAndType(state) // mark right edge, update mode and type
70 |  return true;
```

Algorithm 6: updateModeAndType(*state*)

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