Algorithm 1: structures used

```
struct Node{
    {Boolean,key} markAndKey;
    {Boolean,Boolean,Boolean,NodePtr} child[2];
    Boolean readyToReplace;
};
struct seekRecord{
    NodePtr node; NodePtr parent;
    NodePtr lastUParent; NodePtr lastUNode;
    NodePtr injectionPoint;
};
struct State{
    NodePtr node; NodePtr parent;
    Key key;
    enum mode{ INJECTION, DISCOVERY, CLEANUP };
    enum type{ SIMPLE, COMPLEX };
    seekRecPtr seekRec;
};
```

Algorithm 2: Search(*key*)

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

Algorithm 3: Insert(*key*)

```
while true do
    seek( key, mySeekRec);
    ⟨*, nKey⟩ := mySeekRec→node→markAndKey;
    if key = nKey then return false;
    newNode:= create a new node and initialize its fields;
    which := key < nKey ? LEFT: RIGHT;
    ⟨*,*,*,address⟩ := mySeekRec→injectionPoint;
    out:= CAS(node→child[which],⟨1,0,0,address⟩,⟨0,0,0,newNode⟩);
    if out then return true;
    ⟨*,d,p,address⟩ := node→child[which]; // find out why the CAS failed
    if not (d or p) then continue; // CAS failed due to another insert op
    deepHelp(mySeekRec→lastUNode,mySeekRec→lastUParent);
```

Algorithm 4: Delete(*key*)

```
// initialize the state record
myState→mode:= INJECTION; myState→key:= key;
while true do
    seek( key, mySeekRec);
    node:= mySeekRec→node; parent:= mySeekRec→parent;
    ⟨*, nKey⟩ := node→markAndKey;
    if myState→key≠ nKey then
        // the key does not exist in the tree
        if myState→mode= INJECTION then return false;
        else return true;
    needToHelp:=false;
    // perform appropriate action depending on the mode
    if myState→mode= INJECTION then
        myState→node:= node // store a reference to the node
        out:= inject(myState) // attempt to inject
        if not out then needToHelp:= true;
    // mode would have changed if the op was injected
    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
        if myState→node≠ node then return true;
        myState→parent:= parent // update parent with recent seek
    if myState→mode= DISCOVERY then
        findAndMarkSuccessor(myState);
    if myState→mode= DISCOVERY then
        removeSuccessor(myState);
    if myState→mode= CLEANUP then
        out:= cleanup(myState,0);
        if out then return true;
        else
            ⟨*, nKey⟩ := node→markAndKey; myState→key:= nKey;
            // help if helpee node is not the node of interest
            if mySeekRec→lastUNode≠ node then needToHelp:=true;
    if needToHelp then
    deepHelp(mySeekRec→lastUNode,mySeekRec→lastUParent) ;
```

Algorithm 5: Inject(*state*)

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

Algorithm 6: updateModeAndType(*state*)

```
node:= state→node // retrieve the address from the state record
if node→child[RIGHT] ≠ ⟨*,1,*,*⟩ then // mark right edge if unmarked
    BTS(node→child[RIGHT], DELETE_FLAG);
⟨m,*⟩ := node→markAndKey;
⟨lN,*,*,*⟩ := node→child[LEFT]; ⟨rN,*,*,*⟩ := node→child[RIGHT];
if lN or rN then // update the op mode and type
    if m then
        state→type:= COMPLEX; node→readyToReplace:= true;
    else
        state→type:= SIMPLE; state→mode:= CLEANUP;
else
    state→type:= COMPLEX;
    if readyToReplace then state→mode:= CLEANUP;
    else state→mode:= DISCOVERY;
return ;
```

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

```
// find the smallest key in the subtree rooted at the right child
lastUParent:= node; lastUNode:= right; prev:= node; curr:= right;
while true do
    ⟨n,d,p,left⟩ := curr→child[LEFT];
    if n then break;
    prev:= curr; curr:= left; // traverse the next edge
    if not (d or p) then // keep track of the last unmarked edge
        lastUParent:= prev; lastUNode:= curr
    // update the seek record
    return;
```

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

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

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

```
while true do
    // create two local seek records: cSeek(current seek record) and
       pSeek(previous seek record) used for the traversal
    while true do
        ⟨*,cKey⟩ := curr→markAndKey; // key in the curr of cSeek
        if key = cKey then  // key found; stop the traversal
            done := true; break;
        which := key<cKey ? LEFT: RIGHT;
        ⟨n,d,p,address⟩ := curr→child[which]; // read the next edge
        if n then  // null flag is set; reached a leaf node
            if key stored in anchorNode has not changed then
                done := true; break; // use data from cSeek
            else if anchorNodeof cSeek & pSeek matches then
                done := true; break; // use data from pSeek
            else
                break; // after copying cSeek to pSeek
        if which = RIGHT then  // next edge to be traversed is a right edge
            anchorNode := curr; // keep track of curr node
            anchorKey := cKey; // and its key
        prev := curr; curr: = address; // traverse the next edge
        if not (d or p) then  // keep track the last unmarked edge
            lastUParent := prev; lastUNode := curr;
    if done then
        // initialize the appropriate seek record (cSeek or pSeek)
        return;
```

Algorithm 10: findAndMarkSuccessor(*key*,*seekRec*)

```
// retrieve the addresses from the state record
node:= state→node; seekRec:= state→seekRec;
while true do
    right:= address of the right child;
    findSmallest(node,right,seekRec);
    succNode:= seekRec→node; // retrieve succ node from seek record
    left:= address of the left child of the succNode;
    // try to set the promote flag & copy the node address on the
       left edge using CAS
    out:= CAS(succNode→child[LEFT],⟨1,0,0,left⟩,⟨1,0,1,node⟩);
    if out then break; // promote flag set; promotion will eventually succeed
    // reread the edge to see why the attempt to mark the edge failed
    ⟨n,d,p,left⟩ := succNode→child[LEFT];
    if p then
        if left= node then
            break// successor node has already been selected
        else // the node found is a successor node for another delete operation
            node→readyToReplace:= true
    if not n then continue; // the node found has since gained a left child
    if d then the node found is undergoing deletion. So invoke helping;
updateModeAndType(state); // update the operation mode and type
return;
```

Algorithm 11: removeSuccessor(*state*)

```
// retrieve the addresses from the state record
node:= state→node; seekRec:=state→seekRec;
succNode:= seekRec→node;
if promote flag not set on right child edge of succNode then
    BTS(succNode→child[RIGHT],PROMOTE_FLAG);
node→markAndKey:= ⟨1,succNode→markAndKey⟩; // promote the key
while true do
    succParent:= seekRec→parent; // retrieve parent of the succNode
    right:= right child address of succNode;
    out:= CAS(succParent→child[LEFT],⟨0,0,0,succNode⟩,⟨0,0,0,right⟩);
    if out then break; // successor removed successfully
    // invoke helping if needed
    findSmallest(node, right,seekRec);
    if seekRec→node≠ succNode then break; // successor already removed
node→readyToReplace:= true;
if state→parent≠ null then updateModeAndType(state);
return;
```