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, while we do not allow 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 to see if there is a node in the list with key α which is unmarked.

Procedure Lookup(int key)

```
Data: Node* pred, curr, succ
retry: while true do
    pred = head
    curr = head.next
    while true do
        succ = curr.next
        if curr.next is marked then
            if pred.next.CAS (unmarked curr,unmarked succ) == false then
                go to retry
            end
            curr = succ
        else
            if curr.key ≥ key then
                return <pred,curr>
            end
            pred = curr
            curr = succ
        end
    end
end
```

Shared variables: Node* head

Procedure Insert(int key)

```
Data: Node* pred, curr
      Node node = new Node (key)
while true do
    <pred, curr> = lookup(key)
    if curr.key == key then
        return false
    else
        node.next = unmarked curr
        if pred.next.CAS (unmarked curr, unmarked node) then
            return true
        end
    end
end
```

Procedure Delete(int key)

```
Data: Node* pred, curr, succ
while true do
    <pred, curr> = lookup(key)
    if curr.key != key then
        return false
    else
        succ = curr.next
        if curr.next.CAS (unmarked succ, marked succ) then
            pred.next.CAS (unmarked curr, unmarked succ)
            return true
        end
    end
end
```

Procedure Find(int key)

```
Data: Node* curr = head
while curr.key < key do
    curr = curr.next
end
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 to 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 recovers following a crash, the List data structure is consistent - if it has a pending operation, then either this operation already had a linearization point and affect all other processes, or that it did not affect the data structure at all.