Amplifications & Corrections: Removing Nodes by Looking Down a Structure

T. Reinhardt

Abstract

Periodically, I will push out summaries/capsules of topics that we discuss in Lecture. I generally do this when I feel that points were overlooked or presently in a less than perfectly clear manner—which sometimes happens. Today's topic is removing nodes from linked lists. In particular, I will clarify the algorithm that we discussed in greater detail and I will also provide a minor correction to the reduction logic.

1 The general problem

It's generally easier to add than to remove items from the most commonly implemented data-structures for a number of reasons. First and foremost, we need to ensure that the result of the modification leaves the object in a logically consistent state. For this reason, I prefer to present these algorithms as abstractions that use the structure in question in a natural way.

1.1 Removing all occurrences of an item from a linked-list

When removing *all occurrences* of an `item` from a `LinkedList`, we need to consider the following possibilities:

1. The list is emtpy.
2. The list contains only 1 element.
3. The item in question appears as the first element in the list.
4. The item in question appears after the first element of the list; and, of course,
5. The item does not appear at all.

Of these, the first two seem pretty straightforward. Item number 3, however, is problematic because, if we're not careful, we find ourselves in the impossible position of removing a node that we need in order to get to the remaining nodes in the list. With this in mind, we proposed and presented the following algorithms:

Algorithm 1 Checks that we have any work to do. If so, then solves the "hard problem" first, i.e., removes all occurrences of `item` from the successors. This means that we still need to check the case where the first node contained a `value` that matched the `item`. That is done in the body of Algorithm 1.

Algorithm 2 Recursively examines the *next* node and takes the appropriate action. Note, this algorithm is an example of "looking down" the structure. It avoids the problem of removing an essential node by always operating on the "next." (I also think that I may have omitted the first clause of the "if" statement when I wrote the summary algorithm on the whiteboard, prompting the concern about a non-terminating computation, which is my primary motivation in writing this document!)

Algorithm 1 Removing items from a linked list

```
procedure REMOVE(item, list)
    if list = null then
        return
    end if
    list ← removeAux(item, list)                    ▷ remove from successor links
    if list ≠ null and list.first = item then       ▷ Remove first link?
        list ← list.next
    end if
end procedure
```

Algorithm 2 Removing items that occur on "inner nodes" in a linked-list.

```
function REMOVEAUX(item, list)
    if list = null then        ▷ Ubiquitous but often overlooked in classroom presentation!
        return null
    else if list.next ≠ null and list.next.value = item then
        list.next = removeAux(item, list.next.next)     ▷ Splices out node
    else
        list.next = removeAux(item, list.next)          ▷ otherwise, continues intact
    end if
    return list
end function
```

1.2 Finer points

Obviously, when implementing this in Java you may need to add `return` statements to the bodies of the `if` statements, and you'll need to substitute the appropriate operators for equality tests, etc. Again, I chose a recursive presentation of the algorithm because it's simpler to see what's happening; you may choose to do this iteratively providing that you take the appropriate precautions, etc.